



INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁵ :

H04J 3/24

A1

(11) International Publication Number:

WO 94/07316

(43) International Publication Date:

31 March 1994 (31.03.94)

(21) International Application Number: PCT/US93/08674

(22) International Filing Date: 14 September 1993 (14.09.93)

(30) Priority data:

07/944,682

14 September 1992 (14.09.92) US

(71) Applicant: NETWORK EQUIPMENT TECHNOLOGIES, INC. [US/US]; 800 Saginaw Drive, Redwood City, CA 94063 (US).

(72) Inventors: BURNETTE, John, Lindsay ; 20990 Valley Green Drive, #677, Cupertino, CA 95014 (US). NEWMAN, Peter ; 750 N. Shoreline Blvd., #124, Mountain View, CA 94043 (US).

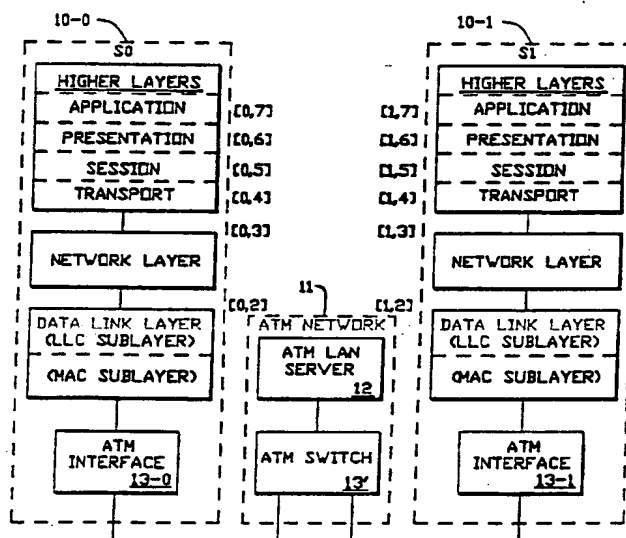
(74) Agent: LOVEJOY, David, E.; Fliesler, Dubb, Meyer and Lovejoy, Four Embarcadero Center, Suite 400, San Francisco, CA 94111-4156 (US).

(81) Designated States: AU, CA, JP, KR, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published

With international search report.

(54) Title: VIRTUAL NETWORK USING ASYNCHRONOUS TRANSFER MODE



(57) Abstract

Asynchronous Transfer Mode Local Area Network (ATM LAN). The ATM LAN is implemented as a set of MAC entities which share a common group address space for the purposes of establishing multicast connections. Each station (10-0) has one or more ATM MAC entities (20-0, 20-1) per physical connection to an ATM network (11). The network ATM LAN service provides the station with ATM LAN configuration information needed for ATM MAC operation. Included in this information is the number of ATM LANs the network has configured for that station.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	FR	France	MR	Mauritania
AU	Australia	GA	Gabon	MW	Malawi
BB	Barbados	GB	United Kingdom	NE	Niger
BE	Belgium	GN	Guinea	NL	Netherlands
BF	Burkina Faso	GR	Greece	NO	Norway
BG	Bulgaria	HU	Hungary	NZ	New Zealand
BJ	Benin	IE	Ireland	PL	Poland
BR	Brazil	IT	Italy	PT	Portugal
BY	Belarus	JP	Japan	RO	Romania
CA	Canada	KP	Democratic People's Republic of Korea	RU	Russian Federation
CF	Central African Republic	KR	Republic of Korea	SD	Sudan
CG	Congo	KZ	Kazakhstan	SE	Sweden
CH	Switzerland	LI	Liechtenstein	SI	Slovenia
CI	Côte d'Ivoire	LK	Sri Lanka	SK	Slovak Republic
CM	Cameroon	LU	Luxembourg	SN	Senegal
CN	China	LV	Latvia	TD	Chad
CS	Czechoslovakia	MC	Monaco	TC	Togo
CZ	Czech Republic	MG	Madagascar	UA	Ukraine
DE	Germany	ML	Mali	US	United States of America
DK	Denmark	MN	Mongolia	UZ	Uzbekistan
ES	Spain			VN	Viet Nam
FI	Finland				

VIRTUAL NETWORK USING ASYNCHRONOUS TRANSFER MODE

BACKGROUND OF THE INVENTION

The present invention relates to networks and particularly to networks of computers that communicate data and other information.

5

Wide Area Networks.

With the increased bandwidth available through transmission channels, for example increases from T1 to T3, and with the increase in bandwidth provided by broadband services such as SONET, larger enterprises are evaluating new applications which require higher speed communications. These new applications will dramatically enhance business productivity, but will require vastly improved network control and management facilities. However, neither private networks nor common carriers have fully addressed the emerging needs of the new communication environment.

10
15Computer Networks

In the computer field, in order for users to have access to more information and to greater resources than those available on a single computer, computers are connected through networks.

In a computer network, computers are separated by distance where the magnitude of the distance has a significant bearing on the nature of communication between computers. The distance can be short, for example, within the same computer housing (internal bus), can be somewhat longer, for example, extending outside the computer housing but within several meters (external bus), can be local, for example, within several hundred meters (local area networks, LANs), within tens of miles (metropolitan area networks, MANs) or can be over long distances, for example, among different cities or different continents (wide area networks, WANs).

20
25
30
35

-2-

Multi-Layer Communication Architecture

For networks, the communication facilities are viewed as a group of layers, where each layer in the group is adapted to interface with one or more adjacent layers in the group. Each layer is responsible for some aspect of the intended communication. The number of layers and the functions of the layers differ from network to network. Each layer offers services to the adjacent layers while isolating those adjacent layers from the details of implementing those services. An interlayer interface exists between each pair of adjacent layers. The interlayer interface defines which operations and services a layer offers to the adjacent layer. Each layer performs a collection of well-defined functions.

Many multi-layered communication architectures exist including Digital Equipment's Digital Network Architecture (DNA), IBM's System Network Architecture (SNA) and the International Standards Organization (ISO) Open System Interface (OSI).

The ISO architecture is representative of multi-level architectures and consists of a 7-layer OSI model having a physical link layer, a data link layer, a network layer, a transport layer, a session layer, a presentation layer, and an application layer.

In the OSI model, the physical layer is for standardizing network connectors and the electrical properties required to transmit binary 1's and 0's as a bit stream. The data link layer breaks the raw bit stream into discrete units and exchanges these units using a data link protocol. The network layer performs routing. The transport layer provides reliable, end-to-end connections to the higher layers. The session layer enhances the transport layer by adding facilities to help recover from crashes and other problems. The presentation layer standardizes

-3-

the way data structures are described and represented. The application layer includes protocol handling needed for file transfer, electronic mail, virtual terminal, network management and other applications.

5 In the n-layer multi-layer models, layers 1, 2, ..., n are assumed to exist in each host computer. Layers 1, 2, ..., n in one host computer appear to communicate with peer layers 1, 2, ..., n, respectively, in another host computer. Specifically,
10 layer 1 appears to communicate with layer 1, layer 2 appears to communicate with layer 2 and so on with layer n appearing to communicate with layer n. The rules and conventions used in communications between the peer layers are collectively known as the peer
15 level protocols. Each layer executes processes unique to that layer and the peer processes in one layer on one computer station appear to communicate with corresponding peer processes in the same layer of another computer station using the peer protocol.

20 Although peer layers appear to communicate directly, typically, no data is directly transferred from layer n on one computer station to layer n on another computer station. Instead, each layer n passes data and control information to the n-1 layer
25 immediately below it in the same computer station, until the lowest layer in that computer is reached. The physical medium through which actual communication occurs from one computer station to another exists below the top layer n and typically below the
30 bottom layer 1.

 In order to provide communication to the top layer n of an n-layer network, a message, M, is produced by a process running in a top layer n of a source computer station. The message is passed from
35 layer n to layer n-1 according to the definition of the layer n/n-1 interface. In one example where n equals 7, layer 6 transforms the message (for exam-

-4-

ple, by text compression), and then passes the new message, M, to the n-2 layer 5 across the layer 5/6 interface. Layer 5, in the 7 layer example, does not modify the message but simply regulates the direction
5 of flow (that is, prevents an incoming message from being handed to layer 6 while layer 6 is busy handing a series of outgoing messages to layer 5).

In many networks, there is no limit to the size of messages accepted by layer 4, but there is a limit
10 imposed by layer 3. Consequently, layer 4 must break up the incoming messages into smaller units, prefixing a header to each unit. The header includes control information, such as sequence numbers, to allow layer 4 on the destination computer to put the
15 pieces back together in the right order if the lower layers do not maintain sequence. In many layers, headers also contain sizes, times and other control fields.

Layer 3 decides which of the outgoing lines to use, attaches its own headers, and passes the data to
20 layer 2. Layer 2 adds not only a header to each piece, but also a trailer, and gives the resulting unit to layer 1 for physical transmission. At the destination computer, the message moves upward, from
25 lower layer 1 to the upper layers, with headers being stripped off as it progresses. None of the headers for layers below n are passed up to layer n.

Virtual Peer To Peer Communication

30 An important distinction exists between the virtual and actual communication and between protocols and interfaces. The peer processes in source layer 4 and the destination layer 4, for example, interpret their layer 4 communication as being
35 "direct" using the layer 4 protocol without recognition that the actual communication transcends down source layers 3, 2, 1 across the physical medium and

-5-

thereafter up destination layers 1, 2, and 3 before arriving at destination layer 4.

The virtual peer process abstraction assumes a model in which each computer station retains control over its domain and its communication facilities within that domain.

Communication Networks Generally

For more than a century, the primary international communication system has been the telephone system originally designed for analog voice transmission. The telephone system (the public switched network) is a circuit switching network because a physical connection is reserved all the way from end to end throughout the duration of a call over the network. The telephone system originally sent all its control information in the 4 kHz voice channel using in-band signaling.

To eliminate problems caused by in-band signaling, in 1976 AT&T installed a packet switching network separate from the main public switched network. This network, called Common Channel Inter-office Signaling (CCIS), runs at 2.4 kbps and was designed to move the signaling traffic out-of-band. With CCIS, when an end office needed to set up a call, it chose a channel on an outgoing trunk of the public switched network. Then it sent a packet on the CCIS network to the next switching office along the chosen route telling which channel had been allocated. The next switching office acting as a CCIS node then chose the next outgoing trunk channel, and reported it on the CCIS network. Thus, the management of the analog connections was done on a separate packet switched network to which the users had no access.

The current telephone system has three distinct components, namely, the analog public switched

-6-

network primarily for voice, CCIS for controlling the voice network, and packet switching networks for data.

5 Future Communication Networks-ISDN

 User demands for improved communication services have led to an international undertaking to replace a major portion of the worldwide telephone system with an advanced digital system by the early part of the twenty-first century. This new system, called ISDN (Integrated Services Digital Network), has as its primary goal the integration of voice and nonvoice services.

 The investment in the current telephone system is so great that ISDN can only be phased in over a period of decades and will necessarily coexist with the present analog system for many years and may be obsolete before completed.

 In terms of the OSI model, ISDN will provide a physical layer onto which layers 2 through 7 of the OSI model can be built.

Telephone Network Domains

 In a telephone network, the system architecture from the perspective of the telephone network is viewed predominantly as a single domain. When communication between two or more callers (whether people or computers) is to occur, the telephone network operates as a single physical layer domain.

30

Communication Network Architectures

 Most wide area networks have a collection of end-users communicating via a subnet where the subnet may utilize multiple point-to-point lines between its nodes or a single common broadcast channel.

35

 In point-to-point channels, the network contains numerous cables or leased telephone lines, each one

-7-

connecting a pair of nodes. If two nodes that do not share a cable are to communicate, they do so indirectly via other nodes. When a message (packet), is sent from one node to another via one or more intermediate nodes, the packet is received at each intermediate node in its entirety, stored there until the required output line is free, and then forwarded. In broadcast channels, a single communication channel is shared by all the computer stations on the network. Packets sent by any computer station are received by all the others. An address field within the packet specifies the intended one or more computer stations. Upon receiving a packet, a computer station checks the address field and if the packet is intended only for some other computer station, it is ignored.

Most local area networks use connectionless protocols using shared medium where, for example, all destination and source information is included in each packet and every packet is routed autonomously with no prior knowledge of the connection required.

In the above-identified application CONCURRENT MULTI-CHANNEL SEGMENTATION AND REASSEMBLY PROCESSORS FOR ASYNCHRONOUS TRANSFER MODE (ATM) an apparatus for concurrently processing packets in an asynchronous transfer mode (ATM) network is described. Packets that are to be transmitted are segmented into a plurality of cells, concurrently for a plurality of channels, and the cells are transmitted over an asynchronous transfer mode (ATM) channel. Cells received from the asynchronous transfer mode (ATM) channel are reassembled into packets concurrently for the plurality of channels.

Accordingly, there is a need for new networks which satisfy the emerging new requirements and which provide broadband circuit switching, fast packet switching, and intelligent network attachments.

- 8 -

SUMMARY OF INVENTION

The present invention is an Asynchronous Transfer Mode Local Area Network (ATM LAN). The ATM LAN is implemented as a set of MAC entities which share a common group address space for the purposes of establishing multicast connections. Each station has one or more ATM MAC entities per physical connection to an ATM network. The network ATM LAN service provides the station with ATM LAN configuration information needed for ATM MAC operation. Included in this information is the number of ATM LANs the network has configured for that station.

In the present invention, a communication system includes an ATM network. The ATM network has a plurality of ports, each port having a unique port address. The ATM network includes one or more ATM switches for connecting sending ports to receiving ports.

The communication system includes a plurality of stations, each station having a unique station address distinguishing the station from other stations. Each station is connected to the ATM network at a port whereby source stations communicate with destination stations. Each station provides packets for transferring information, information including a destination station address, for addressing destination stations. Each station includes a packet converter for converting between packets and cells for transfers between stations.

The communication system provides address resolution for determining a port address corresponding to a destination station address. The address resolution includes multicast for multicasting the destination station address to a group of stations.

The communication system provides management for requesting connections through the ATM

- 9 -

network connecting sending ports to receiving ports whereby packets are transferred from source stations to destination stations by cell transfers through ATM network.

5 ATM LANs may be extended by bridging several ATM LANs together using transparent MAC bridges and routers.

10 Permanent virtual connections or switched virtual connections may underlie the layer management.

15 The communication system operates with a multi-level architecture, such as the ISO architecture, and Logical Link Control (LLC), Media Access Control (MAC) and addressing functions are performed for ATM LANs. An ATM LAN provides support for the LLC sublayer by means of a connectionless MAC sublayer service in a manner consistent with other IEEE 802 local and metropolitan area networks. The ATM LAN interface is built on the user-to-network interface for ATM and adaptation layers.

20 The communication system including the ATM LAN provides the following benefits:

Physical plug-in locations can be moved and changed without changing logical locations.

25 The stations in the communication system are partitionable into multiple work groups.

30 The communication system provides high bandwidth that supports multimedia applications including voice, video, real-time and time-sensitive applications.

 The communication system integrates Wide Area Networks (WAN) and Local Area Networks (LAN) into one system.

35 The foregoing and other objects, features and advantages of the invention will be apparent from the following detailed description in conjunction with the drawings.

-10-

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 depicts a number of user stations connected together in an ATM network system.

5 FIG. 2 depicts the multi-level protocol used to connect two or more stations in the ATM network system of FIG. 1.

FIG. 3 depicts the network layer and the data link layer connected to a ATM interface in the ATM network system of FIGs. 1 and 2.

10 FIG. 4 depicts details of the ATM MAC sublayer and the ATM interface for stations of FIGs. 1 and 2.

FIG. 5 depicts details of the ATM LAN Server and the ATM interfaces of the network of FIGs. 1 and 2.

15 FIG. 6 depicts three ATM LANs configured on a three-switch ATM network.

FIG. 7 is a representation of the details of the ATM MACs of stations S0, S1, S2 and S3 from FIG. 6.

- 11 -

DETAILED DESCRIPTION

In FIG. 1, an ATM network system is shown in which two or more computer stations 10 are interconnected by an ATM network 11 for network communication. The stations 10 include the station S0, S1, ..., Ss designated 10-0, 10-1, ..., 10-s. The ATM network system of FIG. 1 employs, for example, the top six of the seven OSI model layers. The OSI model physical layer 1 is replaced with a ATM interface which operates in an asynchronous transfer mode (ATM) in accordance with the B-ISDN protocol.

In FIG. 2, the ATM network 11 connects, by way of example, the S0 station 10-0 to the S1 station 10-1. The S0 station 10-0 includes the top six OSI layers, namely, the application layer [0, 7], the presentation layer [0, 6], the session layer [0, 5] and the transport layer [0, 4]. The layers 7 through 4 in FIG. 2 are designated as the higher layers and operate in the conventional manner for the OSI model.

In FIG. 2, the S0 station 10-0 includes the network layer [0, 3] and the data link layer, [0, 2]. The data link layer [0, 2] includes the logical link control (LLC) sublayer and the media access control (MAC) sublayer. The MAC sublayer in the data link layer [0, 2] connects to a ATM interface 13-0. The ATM interface 13-0 operates in accordance with the B-ISDN protocol defined by the CCITT.

In FIG. 2, the S1 station 10-1 has the higher layers including the application layer [1, 7], the presentation layer [1, 6], the session layer [1, 5] and the transport layer [1, 4]. The S1 station 10-1 also includes the network layer [1, 3] and the data link layer [1, 2] that connects to the ATM interface 13-1. In FIG. 2, the ATM interface 13-0 for the S0 station 10-0 and the ATM interface 13-1 for the S1 station 10-1 connect to a ATM switch 13' in the ATM network 11. The ATM interfaces 13-0 and 13-1 and ATM switch

-12-

13' operate in accordance with an ATM architecture for ATM communicationn. The ATM LAN communication is under control of an ATM LAN server 12 in the ATM network 11.

5 In FIG. 2 each of the higher layers in the S0 station 10-0 and in the S1 station 10-1 function in a well known manner in accordance with the OSI model. Also, the network layer [0, 3] in the S0 station 10-0 and the network layer [1, 3] in the S1 station 10-1
10 conform to the model OSI The data link layer [0,2] in the S0 station 10-0 and the data link layer [1,2] in the S1 station 10-1 have OSI compatibility. The compatibility with the OSI model at the data link layer enables the ATM network system of FIGs. 1 and
15 2 to be compatible with other local area networks and other networks that conform to the OSI model from layer [2] and above. Below the OSI layer [2], the communication and connections are compatible with the B-ISDN model of the CCITT.

20 The FIG. 2 communication network system is a hybrid of the OSI model above layer [1] and asynchronous transfer mode below the data link layer [2].

 In FIG. 3, further details of the S0 station 10-0 are shown and are typical of all of the other
25 stations 10-1, ..., 10-s of FIG. 1. In FIG. 3, the higher layers 7, 6, 5 and 4 are conventional. Typically the higher layers of the station 10-0 of FIG. 3 are implemented on a processor such as a Sun Workstation.

30 In FIG. 3, the network layer [3] uses any one of a number of standard protocols such as the IP protocol 15, the DEC NET protocol 16, the OSI protocol 17 or the XNS protocol 18. Any other protocol can be implemented in the network layer 3.

35 In FIG. 3, the data link layer [2] includes the LLC sublayer and the MAC sublayer. The LLC sublayer includes the Logical Link Control (LLC) 19 which is

-13-

conventional in the data link layer of the OSI model.

The data link layer [2] also includes the MAC sublayer which as a component of the data link layer [2]. The MAC sublayer typically may include other
5 MAC sublayers in accordance with the standards IEEE 802.3, 802.4, 802.5, 802.6 and FDDI. ATM LANs are, therefore, capable of interoperating with a wide variety of media. ATM LANs interoperate with all IEEE 802 Local Area Networks and Metropolitan Area
10 Networks using transparent bridges and routers. Stations connected to ATM LANs communicate with stations connected to any IEEE 802 LAN or MAN via a bridge.

In accordance with the present invention, the
15 data link layer [2] also includes a new ATM MAC sublayer 22 analogous to the other MAC sublayers 23. The ATM MAC sublayer 22 differs from the other MAC sublayers 23 in that the ATM MAC sublayer 22 communicates with the ATM switch 13 for ATM communication.

20 In FIG. 3, the ATM MAC sublayer 22 includes one or more ATM MACs including, for example, ATM MAC 0, ATM MAC 1, ..., ATM MAC M designated 21-0, 21-1, ..., 21-M respectively. Each of ATM MACs 21-0, 21-1, ..., 21-m defines an ATM local area network (ATM LAN).
25 The ATM MACs of the ATM MAC sublayer 22 connect between the logical link control 19 and the ATM interface 13-0. The control of which of the stations (like the stations 10-0, 10-1, ..., 10-s) are serviced by particular ones of the ATM MACs 21 of FIG. 3 is
30 determined by the station management 20 within the ATM MAC sublayer 22. Other stations (or the same stations) may also be serviced by other local area networks such as Ethernet under control of the other MAC sublayers 23.

35 In FIG. 3, the ATM MAC sublayer is capable of servicing the communication requirements of the stations 10-0 through 10-s of FIG. 1 in one or more

-14-

ATM LANs. Stations can be switched from one ATM LAN to another ATM LAN under control of station management 20 without requirement of modifying the physical connection to the station. For this reason, the ATM
5 LANs are virtual LANs.

In FIG. 4, further details of the ATM MAC sublayer 22 and the ATM interface 13-0 of FIG. 3 are shown.

10 In FIG. 4 the ATM MAC sublayer includes the station management 20 and the ATM MACs including the ATM MAC 0, ..., ATM MAC M designated as 21-0, ..., 21-M.

15 In FIG. 4, the ATM MAC 0 includes the multicast address resolution 24, the unicast address resolution 25, the frame 26 and the connection management 27.

In FIG. 4, the ATM interface 13-0 includes the signaling protocol 28 in the control plane, the ATM ADAPTATION LAYER (AAL) 29, the ATM layer 30 and the physical layer 31.

-15-

1 ATM LANs

1.1 Introduction

 In FIG. 3, the higher layers [7,6,5,5] and [3] are conventional while the data link layer [2] includes the LLC sublayer and the ATM MAC sublayer to implement the Asynchronous Transfer Mode Local Area Networks (ATM LANs). Such an implementation is provided with newly defined Media Access Control (MAC) including addressing protocols. The ATM LAN provides support for the LLC sublayer by means of connectionless MAC sublayer service in a manner consistent with other IEEE 802 local area networks (LAN) and metropolitan area networks (MAN). The ATM LAN interface is built on the user-to-network interface for the ATM layer and the ATM adaptation layer (AAL).

 An ATM LAN includes a set of MAC entities which share a common group address space for the purposes of establishing multicast connections. Each station has one or more ATM MAC entities per physical connection to an ATM network. The network ATM LAN service provides the station with ATM LAN configuration information needed for ATM MAC operation. Included in this information is the number of ATM LANs the network has configured for that station.

 The user-to-network interface at the LLC and MAC levels is defined for the ATM LAN Architecture in a manner analogous to other Data Link Layer architectures.

1.3 ATM LAN Functionality

 An ATM LAN has the following characteristics:

addressing-	all LANs connected by MAC
	bridges use 48 bit addressing
unicast-	all stations can send frames to any other station in the LAN

-16-

duplication- frames are not duplicated
broadcast- all stations can broadcast
to every other station in a LAN
multicast- any station can send to any
5 group address and any station
can register to receive frames
for any group address
promiscuity- any station may chose to
receive all frames with group
10 destination addresses

1.4 ATM LANs

An ATM LAN is a local network having a set of
stations which share a common group address space for
the purpose of establishing multicast connections.
15 An ATM LAN is implemented using services of ATM LAN
MAC, ATM signaling and ATM Adaptation Layers.
Stations may participate in more than one ATM LAN.
ATM LANs may be bridged together using MAC bridges.

ATM LANs are sometimes called Virtual LANs
20 because they are not limited by the limitations of
any physical media characteristics. A single under-
lying ATM network may support many ATM LANs. A
station with a single ATM interface may be connected
to many separate ATM LANs. There are no inherent
25 limitations in the ATM LAN protocol itself to re-
strict either the physical extent or the number of
stations in a particular ATM LAN. Practical limita-
tions, such as multicast traffic, usually limit the
size and scope of ATM LANs.

30 ATM LANs interoperate with a wide variety of
media. ATM LANs can interoperate with all IEEE 802
Local Area Networks and Metropolitan Area Networks
using transparent bridges and routers. Stations
connected to ATM LANs are able to communicate with
35 stations connected to any IEEE 802 LAN/MAN connected
via bridge.

-17-

2 ATM LAN Architecture

2.1 Overview

5 An ATM LAN includes a set of procedures and protocols which work together to provide the services found in IEEE 802 LANs. The AAL and ATM protocols defined by CCITT are augmented by the ATM LAN MAC layer which maps unacknowledged MAC PDUs (MAC Protocol Data Units) onto unacknowledged AAL PDUs trans-
10 mitted over virtual connections provided by the ATM physical layer. The ATM MAC manages connections using an ATM signaling protocol.

2.2 Logical Link Control

15 Stations must comply with 802.2 Type I specification which is defined by ISO 8802. This includes mandatory response to XID (Exchange ID) and Test commands.

When SNAP encapsulations are defined for upper layer protocols they are used.

20 2.3 Station ATM LAN MAC

Each station has one ATM LAN module per physical ATM interface. Each ATM LAN module provides MAC services via one or more ATM MAC entities. The ATM LAN server provides the ATM LAN MAC with config-
25 uration parameters.

2.3.1 ATM MAC Functions

The ATM MAC layer provides the following functions:

30 ATM LAN Configuration- determines the number of ATM LANs which have been configured for the station and the operational parameters needed to establish multicast connections for each ATM LAN.

35 MAC PDU Framing- MAC SDUs (Service Data Units) are encapsulated in

-18-

	Address Resolution-	an AAL specific framing. IEEE 802. 48 bit MAC addresses are mapped onto E.164 ATM addresses.
5	Connection Management-	establishes and rele- ases virtual connections for transmission of MAC PDUs (Protocol Data Units) and reception of frames ad- dressed to registered group (multicast) addresses.
10		
	Multicast Service-	protocol and procedur- es are defined for trans- mission and reception of frames with group address- es. The network provides unreliable delivery via multicast service. The interface to the multicast service is AAL specific. The interface to be used is determined by configuration management.
15		
20		
	2.3.2 ATM MAC Entity Service Interface	
25	The ATM MAC entity provides the following service interface to MAC users.	
	<u>Primitive</u>	<u>Parameters</u>
	M_UNITDATA.request	destination address source address mac service data unit
30	M_UNITDATA.indication	destination address source address mac service data unit
	M_REGISTER_ADDRESS	group address
35	M_UNREGISTER_ADDRESS	group address
	M_REGISTER_ALL	
	M_UNREGISTER_ALL	

- 19 -

2.4 ATM Adaptation Layer

The adaptation layers provide transmission and reception of frames on virtual connections. The standard CCITT AAL are used. In this application, AAL 3 is used to denote AAL 3/4 when end systems negotiate the use of the multiplexing identifier. AAL 4 is used to identify AAL 3/4 when the multiplexing identifiers used are specified by the network. IEEE 802.2 LLC will be identified by a value of 1 in the protocol id field of AAL 3/4 frames.

2.5 ATM Signaling Protocol

The ATM LAN signaling protocol contains a subset of the functions in Q.93B. It provides the following services:

- o establishment of virtual connections (VCs)
- o negotiation of the upper layer protocol (ULP)
- o clearing of connections
- o dynamic port address assignment
- o user to network keep alive

2.6 ATM LAN Server

The ATM LAN server provides configuration and multicast services. It provides operational parameters for each ATM LAN in which each ATM station is configured. Membership in ATM LANs is controlled via policies implemented by the server. These policies may vary between ATM LAN providers. The ATM LAN configuration protocol defines the information provided by stations with which servers may implement policies. Two policies which can be implemented are "port based configuration" and "station based configuration". The ATM LAN server may use the physical cabling to determine LAN membership. This is called "port based configuration". Alternatively, the ATM LAN server may use station MAC addresses to determine LAN membership. This is called "station based configuration". The same station to server protocol is used in either case. The station is not affected

- 20 -

by the configuration policies implemented. When requesting ATM LAN configuration parameters, the station always provides its MAC address(es).

5 The station table shown below is an example of the station-based configuration for the system shown in FIG. 6. The port table shown below is an example of port-based configuration for the system shown in FIG. 6.

10 STATION TABLE
(VLAN MEMBERSHIP)

	<u>VLAN</u>	<u>MAC ADDRESS</u>
	VLAN1	MAC_Add[0] (S0), MAC_Add[1] (S1), MAC_Add[5] (S5) ...
	VLAN2	MAC_Add[2] (S2), MAC_Add[6] (S6) ...
15	VLAN3	MAC_Add[0] (S0), MAC_Add[3] (S3), MAC_Add[4] (S4) ...

PORT TABLE
(VLAN ASSOCIATION)

	<u>Port Addresses [s/p#]</u>	<u>VLAN</u>
20	PA [2,2], PA [2,3], PA [2,4], PA [2,5]	VLAN [3]
	PA [2,6], PA [2,7], PA [2,8], PA [2,9]	VLAN [2]
	.	.
	.	.
25	PA [2,3], PA [3,4]	VLAN [3]
	.	.
	.	.
30	.	.

Each station establishes a VC to an ATM LAN server for each physical interface. A well known group address is used. If redundant ATM LAN servers are providing configuration and multicast service, this service is transparent to the ATM station. The servers agree amongst themselves which ones will serve any particular station. The servers may elect to distribute responsibility for multicast service over several servers. This election is transparent to the station.

3. ATM LAN Configuration Management

A station may belong to one or more distinct ATM LANs. The station will then have been configured with one or more MAC entities each having a unique

-21-

MAC address.

At power-on, the station establishes a VC to the network ATM LAN server. The station ATM MAC sends a configuration enquiry to the ATM LAN server. The enquiry contains the station's MAC address, alan_mac.

```
5      struct alan_req { /* configuration request */
          u_char      alan_proto;
          u_char      alan_pdu_type;
          u_short     alan_seqnum;
10      struct atm_addr alan_mac;
    };
```

Using the unique MAC address, alan_mac, the ATM LAN server determines the number of ATM LANs configured for that station and the configuration for each connected ATM LAN. A configuration response is sent to the station.

```
15      struct alan_config {
          u_char      alan_proto;
          u_char      alan_pdu_type;
          u_short     alan_seqnum;
20      int           alan_num_lans;
          struct alan_parms alan_lan[];
    };
```

The configuration response contains one alans_parms per ATM LAN. For each ATM LAN the configuration manager activates an ATM MAC entity. The parameters in the alan_parms element control the configuration parameters of each ATM LAN "tap".

Each ATM LAN 'tap' is described by the following parameters. The alan_config and alan_update messages contain one or more alan_parms structures.

```
30      struct alan_parms {
          int          alan_version;
          int          alan_aal;
          struct atm_addr alan_port;
          struct atm_addr alan_mcast_base;
35      struct atm_addr alan_lan_uid[];
```

-22-

```
        int                alan_num_mcast;
        u_short            alan_mid;
        u_short            alan_mtu;
    };
```

5 The alan_aal parameter specifies which AAL is used for multicast frames. Currently defined values are 4 and 5 for AALs 4 and 5 respectively. The alan_port is the port address from which VCs are setup for this ATM LAN. The ATM LAN server may specify different port addresses for different taps or may specify the same for all. The ATM MAC entity treats this E.164 address as an unstructured bit string.

10 The ATM LAN manager allocates a range of E.164 group address space for each ATM LAN. The alan_mcast_base is E.164 group address which is used in conjunction with alan_num_mcast (the number of group addresses allocated to the ATM LAN) to map IEEE 802.1 group addresses onto the E.164 group address space. AAL and multicast service parameters are protocol specific.

20 AAL multicast service requires that multicast AAL PDUs be transmitted using multiplexing identifiers, (MIDs), provided by the ATM LAN server. This allows multicast service to be provided via replication functions often found in ATM switch fabrics. Each ATM MAC entity is assigned a LAN unique MID for transmission and must reassemble AAL using the full 10 bit MID.

30 Each ATM LAN is assigned a globally unique identifier, alan_lan_uid. This is a 128-bit name created by the ATM LAN server. The ATM LAN server provides alan_parms structures for the requested MAC addresses. If the station requests configuration parameters for two MAC addresses which belong to the same ATM LAN, two identical alan_parms elements are returned.

35 Once the ATM MAC entities have been created, the configuration manager periodically sends keep alive

-23-

frames on the configuration SVC. If the configuration SVC is released the configuration manager destroys the ATM LAN entities it created. If after some number of retries the ATM LAN server does not respond to keep alive packets, the configuration manager will release the configuration SVC and destroy ATM MAC entities.

Configuration Acquisition Protocol State Machine

10	<u>State</u>	<u>Event</u>	<u>Actions</u>	<u>Newstate</u>
	Inactive	Activate	Setup Request Start timer C1	Wait for Setup Conf
15	Wait for Setup Conf	Release Ind	Setup Request, Start timer C1	Wait for Setup Conf
		Setup Conf	Config Request, Start Timer C2	Wait for Setup Conf
20	Wait for Setup Conf	Timeout	Config Request, Increment Retries	Wait for Setup Conf
25		Max retries	Release, Setup Request	Wait for Setup Conf
		Config Resp	Activate MAC Entities	Active
30	Any state Active	Deactiv- ate	Deactive active MAC entities, Release config- uration VC	Inactive
35		Release Ind	Setup Request, Start timer C1	Setup Request, Start timer C1

4. ATM LAN MAC

The ATM MAC maps IEEE 802.1 flat 48 bit addresses to 60 bit hierarchical E.164 ATM addresses by the address resolution function. Individual IEEE 802. addresses are mapped into port addresses via the ATM Address Resolution Protocol, ATM ARP. Group IEEE 802.1 addresses are mapped to ATM group addresses using a fixed algorithm.

Once an ATM address is determined, the ATM signaling protocol is used to establish a virtual connec-

-24-

tion. The connection is either a unicast connection or a multicast connection depending upon whether the ATM address is an individual or group address. Connection management is responsible for establishing and clearing these connections.

Once the appropriate connection has been determined for a frame, it is encapsulated in an AAL specific encapsulation method. AAL 4 and AAL 5 have distinct multicast mechanisms due to the limitations of AAL 5.

4.1 Framing

4.1.1 AAL 3/4

ATM LAN uses the same MAC framing as 802.6. ATM LANs use 48 bit MAC addresses to enable interoperability with 802 LANs via MAC bridges. As shown in the following table, addresses are encoded as byte quantities as per 802.6.

COM PDU HEAD	MCP HEAD			HEAD EXT	LLC	PAD	CRC 32	COM PDU TRAIL
	DEST ADR	SOUR ADR	MC BITS					
4	8	8	4	0-20	0-9188	0-3	0,4	4

Prot ID	Pad LEN	QOS Delay	Qos Loss	CRC Head Ind Ext Len	Bridging (Not Used)
6	2	3	1	1 3	16

4.2 Addresses

Two types of addresses are used in an ATM LAN, station MAC addresses and ATM (or port) addresses. Both types of addresses may either be individual or group addresses.

MAC station addresses identify individual stations connected to an ATM LAN. Station addresses are 48 bit universally administered 802.1 MAC addresses. These MAC addresses enable interoperability with

-25-

802.1D LAN MAC bridges. Station addresses are used as MAC frame source or destination addresses.

MAC group addresses are used to address frames to multiple destination stations on an ATM LAN. Group addresses are used to set up virtual connections to multiple destination stations without knowledge of those stations' individual addresses. They are used to provide multicast and broadcast services. Broadcast is a specific instance of multicast with all stations receiving frames with well defined group address, specifically all 1's. Group addresses are 48 bit universally or locally administered 802. MAC addresses. The group address with all bits set to one is the broadcast address.

ATM Port addresses or port addresses or ATM individual addresses identify physical ports on switches. They are hierarchical 60 bit E.164 addresses dynamically assigned by the network. Each virtual connection has a port address for at least one endpoint. Port addresses are used in ATM ARP and Signaling PDUs.

ATM group addresses (or multicast port addresses) identify an ATM level multicast group. They are used in signaling PDUs.

25	<u>AddressType</u>			<u>Padding</u>	<u>Address</u>
	ATM port address	110x		no padding	60 bits
30	ATM group address	111x		no padding	60 bits
	MAC station address	1000		12 bits	48 bits
35	MAC group address	1000		12 bits	48 bits

x - indicates whether the address is publicly or privately administered

4.3 Multicast Service

4.3.1 Background

Any station on the LAN can register to receive

-26-

frames addressed to specific group addresses. All stations register to receive frames addressed to group address FFFFFFFFFF which is defined to be the broadcast group address. Any station can send frames to any group address without the knowledge of which stations want to receive them.

4.3.2 ATM LAN Multicast

In an ATM LAN, multicast capability is provided by the multicast server which is part of the LAN server. Stations use that service by establishing virtual connections to the server using the multicast base ATM address provided in the configuration parameters (alan_parms). The multicast base address is a privately administered group E.164 address. Virtual connections with a group ATM address at one endpoint are multicast VCs. When setting up a multicast VC the station may request transmit only access so that it will not receive frames transmitted on that VC.

IEEE 802.1 48 bit addressing provides for up to 2^{46} possible group addresses all registered by various stations in one LAN. Few ATM networks could support 2^{46} virtual connections. To bridge this gap in service offering and network capability, each ATM LAN is configured to support a small (typically 100s) number of multicast circuits. This number is exported in the alan_parms configuration element. Each ATM MAC entity is also provided with a multicast base address which is treated as a 64-bit integer. These two numbers are used to map many 48-bit IEEE group addresses to fewer ATM group addresses which are then used to setup multicast connections. If alan_num_multicast is zero, then the 48-bit group address is added to alan_mcast_base. Otherwise the 48-bit group address is treated as a 16 most significant bits of the 48-bit group address are Exclusive-Ored into the 32 least-significant bits, the result

-27-

is divided by `alan_num_mcast` and the resulting remainder is added to `alan_mcast_base`. In either case, the result value is used as a group address to set up a multicast connection for that group address.

5 4.3.3 Registering for a group address

Each ATM MAC entity maintains a list of group addresses for which its users have requested it receive frames. Each of these group addresses is mapped onto a ATM group address when the MAC entity is given its `alan_parms` information, that is, when it becomes active. There after, the ATM MAC entity will maintain a multicast connection for each port address derived from the above computations. Note, several MAC group addresses may map onto one group port address. In this case, only one connection is maintained for those MAC group addresses. If the network releases a multicast connection, the ATM MAC entity will re-establish another one.

20 The ATM MAC entity will always maintain a multicast connection for the group port address derived from the broadcast MAC address.

4.3.4 Transmission of Multicast MAC PDUs

When an ATM MAC entity is presented with a `M_UNITDATA.request` with a group destination address it maps the group MAC address to the group ATM address, and transmits the MAC PDU on the connection established to that port address. If no connection is already established, the frame is queued until one is established. Multicast connections setup solely for the transmission of multicast PDUs are aged in the same fashion as those setup for unicast PDUs.

30 4.3.5 Reception of Multicast MAC PDUs

The group destination addresses in received MAC PDUs are checked against the list of registered group addresses. If the group addresses are not registered, the frame is dropped. This dropping is necessary because transmitters may map MAC group

- 22 -

addresses onto a multicast connection established to register other group addresses.

5 All group addressed frames are not received on corresponding multicast connections. Stations listening for multicast frames must be prepared to receive those frames on either the appropriate multicast VC or the broadcast VC.

4.3.6 Unregistering a group address

10 Multicast connections established for registered group addresses are not aged. They are not released until the last MAC service users want to receive frames addressed to any of the group addresses mapped onto that connection.

15 The ATM MAC entity maintains reference counts on the number of MAC service users which have registered a group address. A reference count on the multicast connection is maintained for each MAC group which maps onto the connections group ATM address.

4.3.7 AAL 4 Multicast Service

20 Stations connected to multicast VCs can receive frames from many sources simultaneously. The multiplexing identifier (MID) in the ALL4 SAR header is used to correctly reassemble these frames. MIDs are unique within a given ATM LAN. The LAN server
25 assigns a unique MID to each port address.

Up to 1023 stations may be connected to an ATM AAL 3/4 LAN. Each station has a globally unique 48-bit address per ATM LAN. Each station is assigned one MID per ATM LAN (local port address to the
30 station) to be used when transmitting frames on multicast VCs. Stations may not transmit more than one frame simultaneously on multicast VCs with the same local port address. Each station implements MAC level address filtering for frames received on
35 multicast VCs.

Each station has a multicast filter which is used to filter frames received on broadcast VCs. This

-29-

filter may be implemented in hardware or software. The filter is necessary because each ATM network provides limited multicast service and stations may broadcast unicast frames.

5 4.3.8 AAL 5 Multicast Service

AAL 5 does not provide for multiplexing frames on a single VC simultaneously. The mid field in the alan_parms structure is ignored. There is no limit on the number of stations (or ATM MACs) which may belong to an AAL 5 ATM LAN.

10

4.4 ATM Address Resolution Protocol

Individual IEEE 802.1 MAC addresses are mapped into port addresses via the ATM Address Resolution Protocol (ATM ARP). Once the port address is determined the ATM signaling protocol is used to establish a virtual connection.

15

4.4.1 ATM ARP Operation

Stations connected directly to ATM LANs will, conceptually, have address translation tables to map MAC addresses (both station and group addresses) into virtual connection identifiers. The MAC-to-port table, provides mappings from MAC addresses to port addresses.

20

The MAC transmission function accesses this table to get next hop port address given destination station address. This table is updated when new station address to port address mappings are learned via ATM ARP and when MAC group address to ATM group address mappings are computed. The entries in the MAC to port table are updated when ATM ARP requests and replies are received.

25

30

When the MAC layer is presented with a frame for transmission, it looks up the destination address in the station to port address table. If an entry is found, connection management selects the appropriate virtual connection upon which the frame should be transmitted.

35

-30-

If no entry is found, a new entry is allocated for that MAC address. If the MAC address is a group address, an ATM group address is computed using an AAL specific function. This operation permits the broadcast VC to be established without sending ATM ARP requests. Mapping individual MAC addresses to port addresses is accomplished by broadcasting an ATM ARP request for the MAC addresses to all stations connected to the ATM LAN. The ATM ARP request carries the senders MAC and port address mapping. All stations receive the request. The station with the specified MAC address responds with an ATM ARP reply. The responder updates its MAC-to-port table using the information in the request. The reply carries both the responders' and the requestors MAC and port addresses. When the requestor receives the ATM ARP reply, it updates its port-to-MAC address table.

MAC to Port entry

<u>Station Address 48</u>	<u>Next Hop Port</u>	<u>Status</u>
<u>Bit 802.1 MAC</u>	<u>Address E.164</u>	

The requesting station must transmit MAC frames on broadcast connections until it receives responses to its ATM ARP requests. It may then set up a connection using the port address in the reply. Usually, the responder sets up the connection before replying.

The ATM ARP function times out entries in the MAC-to-port table when they have been idle for some time. Connection management is notified when entries in the MAC-to-port table are added, updated or deleted. Connection management notifies ATM ARP when connections are established and released. Entries in this table are deleted when an SVC establishment to the port address fails. They are deleted when the connection corresponding to an entry is released.

- 31 -

TABLE 5: Port to VPI-VCI entry

<u>Local Port</u> <u>Address</u>	<u>Peer Port</u> <u>Address</u>	<u>OOS</u>	<u>VPI/VCI</u>
-------------------------------------	------------------------------------	------------	----------------

5

4.4.2 ATM ARP PDUs

ATM ARP requests and replies are encapsulated in 802.2 LLC and the appropriate AAL for the connection upon which they are sent. ATM ARP requests are always broadcast. Therefore they are encapsulated in the AAL used for multicast connections. ATM ARP replies are usually sent on point to point connections. The ATM MACs negotiate the AAL to be used for that connection. The reply is then encapsulated in 802.2 LLC and the specific AAL framing.

15

The ATM ARP messages are:

/*

* ATM Address Resolution Protocol.

*/

20

```
struct atm_arp {
    u_short    aa_llp; /* lower layer protocol */
    u_short    aa_ulp; /* upper layer protocol */
    u_char     aa_llp_len;
    u_char     aa_ulp_len;
    u_short    aa_msg_type;
    u_char     aa_sender_port[8];
    u_char     aa_sender_mac[6];
    u_char     aa_target_port[8];
    u_char     aa_target_mac[6];
};
```

30

/* aa_msg_type's */

#define ATM_ARP_REQUEST 1

#define ATM_ARP_REPLY 2

35

The aa_ulp_len and aa_llp_len fields are always 6 and 8 respectively. The aa_ulp field is set to 16. The sender mac and port addresses are set to the sender's Mac and Port addresses for request and non-proxy reply messages. The aa_send_mac field in

- 32 -

proxy replies contains the aa_target_mac from the corresponding request. The aa_target_mac is always set to the Mac address needing resolution in requests and it is set to the requestor's Mac address in
5 replies. The aa_target_port is undefined in requests and in replies it contains the aa_sender_port from the corresponding request. The recipient of a reply verifies that the aa_target_port corresponds to one of its own port addresses.

10 4.5 Connection Management

Once a MAC address has been resolved to a ATM address a connection to the station receiving frames for that MAC address can be set up and those frames can be transmitted directly to that station rather
15 than broadcast. Connection management is responsible for defining the connection establishment and release policies. The ATM signaling protocol is used to establish connections for ATM LAN MAC frames. A specific upper layer protocol identifier is reserved
20 for ATM LAN MAC frames.

4.5.1 Connect Establishment

Connections are established when an Unacknowledged Data Request needs to be transmitted to a MAC address for which a MAC-to-ATM address mapping is
25 known, but no connection to that ATM address, is established (or emerging). It is possible for two MAC entities to simultaneously establish connections to each other. When connection management receives connection setup SDU from ATM signaling, it checks to
30 see if a connection to the peer port address already exists. If another connection exists (or is being established), the connection initiated from the lower port address is released. Thus there will never be more than one connection established between two ATM
35 MAC entities.

While a connection is being setup, frames which would be transmitted on that connection once it is

- 33 -

established must be queued or dropped. Frames should not be broadcast. At least one frame must be queued. Implementations may chose to queue more. Once the connection is set up, any queued frames are transmitted. The first frame transmitted on a connection initiated by a station must be the ATM ARP response for the an ATM ARP request.

4.5.2 Quality of Service

Currently, distinct qualities of service may be defined for ATM MAC PDUs.

4.5.3 Connection Release

Connections for which there is no MAC-to-ATM address mapping are held for the product of the number of ATM ARP retries and retry interval and then released. The MAC-to-ATM address mappings are aged separately.

When ATM ARP deletes all the translations to a specific ATM address, all connections to that ATM address are released.

When a connection is released, the ATM ARP function deletes all MAC to ATM translations for that connection's remote ATM address.

4.6 Frame Reception

Frame Reception Stations are responsible for performing filtering of incoming frames. Unicast addressed frames for other stations will be received on the broadcast VC. Multicast frames for unregistered multicast addresses may be received on multicast VCs. These frames are not passed up to the MAC service user.

4.7 Address Resolution and Connection Establishment Example

In this example, the steps are described that are required for one station, called Lyra, to deliver a MAC UNITDATA SDU to another station, called Altera, assuming neither station has had any prior communication. It is assumed that both stations are part of the same ATM LAN. These steps are only required for

-34-

- the initial transmission from Lyra to Altera. Additional MAC PDUs may be transmitted on the connection setup by these steps until either station decides it no longer wishes to maintain the connection. In this example, MAC addresses are expressed in xx:xx:xx:xx:xx:xx form where each pair of hex digits, xx, is one octet for the address. Port addressees are expressed in the same form except that they have 8 octets.
- 10
- o An ATM MAC service user on Lyra provides the ATM MAC with an UNITDATA SDU to be sent to station address 00:80:b2:e0:00:60. The MAC consults its MAC to port address table, but finds no translation.
- 15
- o The MAC creates an ATM ARP request for MAC address 00:80:b2:e0:00:60. The request contains Lyra's own MAC and port addresses, 00:80:b2:e0:00:50 and d1:41:57:80:77:68:00:02 respectively. The ATM ARP is encapsulated in LLC/SNAP. The destination MAC address is ff:ff:ff:ff:ff:ff (the broadcast address). The ATM MAC recursively invokes itself to transmit the ATM ARP request.
- 20
- 25
- o The MAC address to port address table is searched for the broadcast MAC address and the corresponding port address is obtained, f1:41:57:80:77:68:01:01. The station established a connection to this port address when the ATM LAN MAC entered the active state. The ATM ARP PDU is encapsulated in an 802.6 frame and passed to the AAL 4 function along with the MID associated with this ATM MAC entity for transmission of that multicast connection.
- 30
- 35
- o The MAC must transmit the MAC SDU. In lieu of a valid MAC address to port address mapping the broadcast MAC to port mapping and associated connection

-35-

are used. The MAC SDU is encapsulated in an 802.6 frame and passed to the AAL 4 function with the MID associated with this ATM MAC entity for transmission of that multicast connection.

5 All the above took place on Lyra. The subsequent steps take place on Altera as it receives the ATM ARP and the ATM MAC PDU containing user data.

10 o The ATM ARP is received by all MAC entities including Altera. The other MACs determine that the requested MAC address is not theirs and ignore the request. Altera determines that its MAC address is in the request. Altera updates its MAC to port address table with Lyra's MAC and port addresses provided in
15 the ATM ARP request. Next an ATM ARP reply is constructed using Altera's port and MAC addresses. This request, in the form of an MAC SDU with Lyra's MAC address as the destination, is passed to the ATM MAC entity.

20 o The ATM MAC looks up Lyra's MAC address in the MAC to port address table. It finds Lyra's port address. The port to VCI table is searched using that port address. No entry is found. Connection management is invoked to establish a connection to Lyra.
25 Connection management passes a SETUP request to ATM signaling. The MAC queues the ATM ARP response until the connection is established.

30 o Altera ATM signaling module sends a SETUP PDU to establish a connection to port address d1:41:57:80:-77:68:00:02. The upper layer protocol (sometimes called upper layer compatibility) is the ATM LAN MAC. (This is not a function of the ATM MAC. But it is
35 included for illustrative purposes.)

o Next all stations receive the MAC SDU containing

-36-

the user data on the broadcast connection. All stations except Altera determine that the destination MAC address is not theirs and drop the frame. Altera accepts the frame strips off the 802.6 and LLC/SNAP overhead and passes the frame up to the user function identified by LLC/SNAP.

At this time, the SDU provided to Lyra's ATM MAC has been delivered to the appropriate MAC user on Altera. However, the MAC entities continue connection establishment and address resolution for subsequent communications between the two stations. The next sequence of operations occurs on Lyra.

o ATM signaling on Lyra receives a connection setup indication from the network. This indication is passed up to the upper layer protocol which in this instance is the ATM MAC.

o The ATM MAC receives a setup indication SDU from signaling. At this point Lyra knows some other station's ATM MAC is trying to setup a connection to it. The port to vci table is searched for a connection to the callers port address. In this case none is found. The connection is accepted by passing a CONNECT SDU to ATM signaling. The MAC starts an idle timer for the connection. Note, that the ATM MAC can not use this connection until an ATM ARP request or response is received indicating MAC addresses for stations accessible via the connection.

o Lyra's ATM signaling transmits a CONNECT PDU to the network. Typically, network communication is bi-directional. Assuming this is the case the MAC service user on Altera has responded to the MAC SDU indication with a MAC SDU request. The following actions take place on Altera. The ordering of the arrival of MAC SDU and the CONNECT SDU are arbitrary.

-37-

- 5 o The MAC service user passed the ATM MAC an SDU with a destination MAC address of 00:80:b2:e0:00:50 (Lyra's). The MAC finds the mapping from MAC address to port address learned when the ATM ARP request was received from Lyra. The MAC next finds that it is setting up a connection to Lyra's port address and that the connection is not yet established. A MAC PDU is created from the MAC SDU and queued waiting connection establishment.
- 10 o Altera ATM signaling receives a connect PDU. This is passed up to the MAC as a SETUP confirmation. The ATM signaling sends a CONNECT acknowledge PDU to Lyra. The connection is considered established.
- 15 o Altera's ATM MAC, upon receiving the SETUP confirmation, transmits all frames which were queued awaiting connection establishment. The ATM ARP reply is the first frame to transmitted. It is followed by
- 20 the MAC PDU containing user data.
- At this TIME, address resolution and connection are complete on Altera. Any further frames addressed to Lyra's MAC address will use the new connection. The connection is not established on Lyra. Also Lyra
- 25 still does not have a mapping for Altera's MAC address. The following actions complete address resolution and connection establishment on Altera.
- o The ATM ARP reply is received on the connection
- 30 which is still being setup. (Note most ATM networks have slower signaling channels than payload channels. Typically the ATM ARP response will be received prior to the CONNECT acknowledge PDU.)
- 35 o The MAC enters Lyra's MAC address to port address mapping in the MAC to port table. At this point any MAC UNIT-DATA requests will be queued until

- 38 -

the SETUP complete indication for the connection is passed up from ATM signaling.

5 o The MAC PDU containing user data from Lyra's MAC users is received. The 802.6, LLC and SNAP headers are removed and a MAC UNITDATA indication is passed up to the appropriate MAC service user.

10 o Altera's ATM signaling receives a CONNECT_ACK PDU. This moves the connection into established state. The ATM signaling function passes up a SETUP COMPLETE indication informing the ATM MAC it may transmit on the connection. Connection management starts its idle timer for the connection.

15 The connection is now established on both stations. One or more MAC UNITDATA SDUs have been delivered. The connection will be timed out as per local policy decisions.

20 ATM LAN Code Overview

One detailed embodiment of computer software code used in connection with the present invention appears in the VIRTUAL NETWORK USING ASYNCHRONOUS TRANSFER MODE APPENDIX.

25 The ATM LAN MAC code in the appendix is organized by functional components and Operating System (OS) dependencies. The file if_atm.c contains the routines which contain OS dependencies and which are typically implemented differently for each OS. The
30 unicast unit 25 and multicast unit 24 address resolution functions are implemented in the file atmarp.c. The file atmarp.h contains the definitions for the ATM ARP protocol and the structures used by atmarp.c to implement the protocol. The file atm.c implements
35 the function of connection management unit 27. Those routines interact with the ATM signaling function to establish and release connections. The framing unit

-39-

26 function is implemented in the OS specific file
if_niu.c in the routines niuoutput(), atm_mac_input()
which encapsulate and decapsulate frames respective-
ly. The station management unit 28 functions are
5 implmenented in atm_init.c and in parts of the ATM
signaling unit 28 in the files svc.c, svc_utl.c and
svc_pdu.c. The ATM LAN server unit 12 functions are
implemented in the files lm.c, lm_cfg.c, lm_mgmt.c
and lm_util.c.

10 In the APPENDIX the configuration management
units 20 and 40 are implemented in an alternate
embodiment wherein the operational unit 28 PDUs
rather than in a switched VCs as previously describe-
d.

15 While the invention has been particularly shown
and described with reference to preferred embodiments
thereof it will be understood by those skilled in the
art that various changes in form and details may be
made therein without departing from the spirit and
20 scope of the invention.

40
atm.c
-2-

atm.c

```
/*
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */
```

```
/*
 * This file contains: atm_init() is called at initialization.
 * atm_find_atp() returns an atmif given a MAC and physical I/f.
 * atm_find_mac() returns an atmif given a MAC. atm_sdu_handler() is
 * the interface to ATM signaling. atm_release() releases a VC and
 * frees any queued packets. atm_find_at() searches for an arptab
 * entry given a mac address. atm_initiate_setup() initiates VC
 * establishment.
 */
```

```
static char    sccsid[] = "%A%";
```

```
#include "atm.h"
#include "svc.h"
#include "debug.h"
#include "nlu.h"
#include "atmarp.h"
#include "llc.h"
#include "if_atm.h"
```

```
int            atm_trace = 2;
#define TL1    1
#define TL2    atm_trace > 1
#define TL3    atm_trace > 2
#define TL4    atm_trace > 3
#define TL5    atm_trace > 4
```

```
int            atm_assert_panic = 1;
```

```
#define HASHMA(x) HASH_LOW((x)->aa long[1]) /* pass in a atm_addr */
#define HASH_LOW(part0) (((part0 > 8)^(part0))&0xf) /* pass in low 32 bits
 * of addr */
```

```
/*
 * atm_init() is called to allocate atm_glob which contains all the
 * ATM LAN MAC global variables which are written after program load.
 */
```

```
atm_init()
{
    int            i, atm_mac_input(), atm_sdu_handler();
    struct atm_globs *ag = atm_glob;

    if (ag->atm_initialized)
```

41

atm.c

-3-

```

    return 0;
    ag->atmifn = NNIU * NATMS;
    ag->atm_ulp = ulp_register(LMI_MAC_ORG, LMI_MAC_PID,
        atm_mac_input, atm_sdu_handler, 0);
    ((u_short *) &ag->llc_def)[0] = 0xaaaa;
    ((u_short *) &ag->llc_def)[1] = 0x0300;
    ((u_short *) &ag->llc_def)[2] = 0x0;
    ((u_short *) &ag->llc_def)[3] = 0x0;
    ag->atm_null.aa_long[0] = 0;
    ag->atm_null.aa_long[1] = 0;
    ag->atm_null.aa_type = AAT_NULL;
    ag->atm_broadcast.aa_long[0] = 0;
    ag->atm_broadcast.aa_long[1] = 0;
    ag->atm_broadcast.aa_type = AAT_MAC;
    for (i = 0; i < 6; i++)
        ag->atm_broadcast.aa_byte[ATM_FIRST_MAC + i] =
            (u_char) 0xff;
    ag->atm_initialized = 1;
    return;
}

/*
 * atm_find_atp() returns an ATM LAN structure pointer, atp, given a
 * port address and a phys i/f.
 */
struct atmif *
atm_find_atp(pc, port)
    struct pcif *pc;
    struct atm_addr *port;
{
    struct atmif *atp;

    for (atp = pc->pc_atmif; atp; atp = atp->ati_next)
        if (ATM_ADDR_EQ(atp->ati_port, *port))
            return atp;
    return (struct atmif *) 0;
}

/*
 * atm_fint_at() returns a pointer to an atm interface entry to be
 * used for a specific MAC address.
 */
struct atmif *
atm_find_mac(mac)
    u_char *mac;
{
    struct atm_addr addr;
    struct pcif *pc;
    struct atmif *atp;

```

42
atm.c
4

```

atm_bzero(&addr, sizeof(addr));
atm_bcopy(mac, &addr.aa_byte[2], 6);
addr.aa_type = AAT_MAC;

for (pc = svc_glob->svc_pcif; pc < svc_glob->svc_pcifn; pc++)
    for (atp = pc->pc_atmif; atp; atp = atp->atp_next)
        if (ATM_ADDR_EQ(atp->atp_mac, addr))
            return atp;
return 0;
}
/*
 * atm_sdu_handler() handles signaling SDUs from the ATM signaling
 * module. The necessary ATM ARP routines are called at connection
 * establishment and release.
 */
atm_sdu_handler(vp, sdu, len)
    struct vcte *vp;
    struct setup *sdu;
    int len;
{
    struct ulptab *ulp;
    int rtn;
    struct vcte *ovp; /* other VC */

    ASSERT(VALID_VP(vp));
    TR1(TL2, "atm_sdu_handler(%s)\n",
        svc_xdu_type_str(sdu->lmi_pdu_type));

    if (vp->vcte_flags & VCTEF_MCAST_SERVER) {
        if (ulp = ulp_find(LMI_MCAST_PID, LMI_MCAST_ORG)) {
            ASSERT(VALID_ULP(vp->vcte_ulp));
            ulp_free(vp->vcte_ulp);
            ulp_tax(ulp);
            vp->vcte_ulp = ulp;
            (*ulp->ulp_lmi) (vp, sdu, len);
        } else
            atm_release(vp, INVALID_DST_ADDR);
        return;
    }
    switch (sdu->lmi_pdu_type) {
    case SDU_SETUP_IND:
        if (lsvc_find_local_port(vp->vcte_pcif,
            &((struct setup *) sdu)->lmi_callee)) {
            atm_release(vp, INVALID_DST_ADDR);
            break;
        }
        ASSERT(vp->vcte_atmif == 0);
        vp->vcte_atmif = atm_find_atp(vp->vcte_pcif,
            &vp->vcte_local);
        ASSERT(VALID_ATP(vp->vcte_atmif));
    }
}

```

43
atm.c
-5-

```

ovp = svc_find_vc(vp->vcte_pcif, &vp->vcte_local,
    &vp->vcte_peer, atm_glob->atm_ulp,
    VCS_NOT_DEAD_OR_DYING & ~(1 << VCS_WSR));
if (ovp && bcmp(&vp->vcte_local,
    vp->vcte_peer, sizeof(struct atm_addr))) {
    if (atm_incoming_vc_is_better(vp, ovp)) {
        arp_release(ovp->vcte_atmif->ati_arptab, ovp);
        atm_release(ovp, VC_REDUNDANT);
    } else {
        atm_free_msg(sdu);
        atm_release(vp, VC_REDUNDANT);
        break;
    }
}
sdu->imi_pdu_type = SDU_SETUP_RESP;
if (rtn = svc_sdu(vp->vcte_pcif, vp, sdu,
    sizeof(struct setup)))
    TR1(TL2, "svc_sdu(SETUP_RESP)->%d\n", rtn);
break;
case SDU_SETUP_COMP:
    if (bcmp(&vp->vcte_local, vp->vcte_peer,
        sizeof(struct atm_addr)))
        arp_setup(vp->vcte_atmif->ati_arptab, vp);
case SDU_SETUP_CONF:
    atm_free_msg(sdu);
    if (vp->vcte_packet)
        atm_send_packets(vp);
    break;
case SDU_RELEASE_IND:
    if (vp->vcte_packet)
        atm_free_packets(vp);
    arp_release(vp->vcte_atmif->ati_arptab, vp);
    atm_free_msg(sdu);
    break;
case SDU_STATUS_RESP:
    atm_free_msg(sdu);
    break;
default:
    panic("unknown sdu");
    break;
}
}

/*
 * atm_incoming_vc_is_better() choses the better VC given two
 * redundant VCS. We limit the number for VCs to one between every
 * pair of ports. When a new VC is initiated we check for a
 * duplicates. The VC Initiated by the station with the lowest port
 * address is released. It is imperative that both sides use this
 * algorithm otherwise we could end up in a deadly embrace.

```

44
atm.c
6

```

*/
atm_incoming_vc_is_better(ivp, ovp)
struct vcte *ivp; /* incoming VC */
struct vcte *ovp; /* outgoing VC (we initiated) */
{
    struct atm_addr *ours, *theirs;
    int true;

    ASSERT(ovp != ivp);
    TR2(TL3, "atm_sdu_handler: dup vcs found %x & %x\n", ivp, ovp);
    if (ivp->vcte_pcif->pc_flags & PCIF_NIU_TO_NIU) {
        /* compare MAC addresses */
        true = ivp->vcte_pcif->pc_flags & PCIF_OTHER_MAC_ADDR_IS_HIGHER;
    } else { /* compare port addresses */
        true = ATM_ADDR_GT(ivp->vcte_peer, ivp->vcte_local);
    }
    TR2(TL3, "atm_incoming_vc_is%s better(%s) \n",
        true ? "" : "not", e160_ntoa(&ivp->vcte_peer));
    ASSERT(ovp); /* keep ovp alive */
    return true;
}

```

```

/*
 * atm_release() sends a release_req to the svc module and frees any
 * queued packets.
 */
atm_release(vp, cause)
struct vcte *vp;
{
    struct release *rdu;
    int rtn;

    if (vp->vcte_packet)
        atm_free_packets(vp);

    if (!(rdu = (struct release *) atm_alloc_msg())) {
        printf("atm_release: no memory");
        return;
    }
    rdu->lmi_proto = LMI_PROTOCOL;
    rdu->lmi_pdu_type = SDU_RELEASE_REQ;
    rdu->lmi_cref_type = vp->vcte_cref_type;
    rdu->lmi_cref_value = vp->vcte_cref_value;
    LMI_SET_ELEMENT(&rdu->lmi_cause, LMI_RELEASE_CAUSE, cause);
    if (rtn = svc_sdu(vp->vcte_pcif, vp, rdu, sizeof(*rdu)))
        TR1(TL1, "atm_release: release failed %d\n", rtn);
}
/*

```

45
atm.c
-7-

* atm_find_at() searches for an arptab entry given a mac address.
 * The "best" entry upon which to send to the mac address is
 * returned. If the entry has no VC this routine attempts to setup
 * one.

*/

u_char atm_macbroadcastaddr[6] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff};

struct aate *

atm_find_at(atp, dst)

struct atmif *atp;

u_char *dst;

{

struct vcte *vp = 0;

struct aate *at;

ASSERT(VALID_ATP(atp));

at = atm_mac_to_aate(atp, dst);

if (!at) /* we queue frames now... ||

* lat->aate_vcte) */

at = atm_mac_to_aate(atp, atm_macbroadcastaddr);

if (!at)

return 0;

if (!at->aate_vcte)

atm_initiate_setup(atp, at);

return at;

}

/*

* atm_initiate_setup() initiates a vc setup between to the port
 * addresses using the specific interface. If there is already a vc
 * coming up between the two ports using that interface we do not
 * bother. This huarantees that we do not initiate two VCs to the
 * same port address. When we get setup indications we must also
 * check for duplicates and decide which VC to keep.

*/

atm_initiate_setup(atp, at)

struct atmif *atp;

struct aate *at;

{

int rtn;

struct pcif *pc = atp->ati_pcif;

struct atm_addr *from = &atp->ati_port;

struct vcte *vp;

struct setup *pdu; /* setup pdu and sdu are the same */

struct lmi_ulp *lu;

if (pc->pc_sig->vcte_state != VCS_ACTIVE)

4
atm.c
-8-

```

/* no signaling yet */
return 0;
vp = svc_find_vc(pc, from, &at->aate_atmaddr,
    atm_glob->atm_ulp, VCS_NOT_DEAD_OR_DYING);
if (vp)
    goto found_a_vc;

pdu = (struct setup *) atm_alloc_msg();
if (!pdu)
    return 0;
pdu->lmi_proto = LMI_PROTOCOL;
pdu->lmi_ncalls = 1;
pdu->lmi_caller = *from;
pdu->lmi_callee = at->aate_atmaddr;
pdu->lmi_pdu_type = SDU_SETUP_REQ;
pdu->lmi_cref_type = LMI_CREFTYPE_SVC;
pdu->lmi_cref_value = 0; /* let svc module pick one */
lu = (struct lmi_ulp *) &pdu[1];
lu->af_type = LMI_ULP;
lu->af_aal = PAYLOAD_AAL_4;
lu->af_pid = LMI_MAC_PID;
lu->af_org = LMI_MAC_ORG;

if (rtn = svc_sdu(pc, 0, pdu, sizeof(*pdu) + sizeof(*lu)))
    TR1(TL2, "atm initiate setup: svc sdu->%d\n", rtn);
vp = svc_find_vc(pc, from, &at->aate_atmaddr,
    atm_glob->atm_ulp, VCS_NOT_DEAD_OR_DYING);
found_a_vc:
if (vp) {
    at->aate_vcte = vp;
    vp->vcte_atmif = atp;
    svc_inc(vp);
    TR1(TL1, "atm find_at: setup to %s failed\n",
        e160_ntoa(&at->aate_atmaddr));
}
}

```

47
atm.h
-8-

```

/* atm.h
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */

#ifndef NIU_ATM_H
#define NIU_ATM_H included

#include "bytes.h"
#include "unipdu.h"

/*
 * atm mac service interface (asi). This is the same as an ethernet
 * header so that upper layers can simply assume ATM is an ethernet.
 */
struct atmmsl {
    u_char    asi_dst[6];
    u_char    asi_src[6];
    u_short   asi_type;
};

/*
 * Structure of an ATM mac header for aal type 4, this is an 802.6
 * header.
 */

struct atm_header {
    struct atm_addr atm_dst;
    struct atm_addr atm_src;
    union {
        struct {
            u_int    mcb_pid:6;
            u_int    mcb_pad:2;
            u_int    mcb_delay:3;
            u_int    mcb_loss:1;
            u_int    mcb_crc:1;
            u_int    mcb_elen:3;
            u_int    mcb_pad1:16;
        }
        u_int    mcbits;
        u_int    atm_mcb_long;
    }
    un_mcb;
};

#define atm_mcbits un_mcb.atm_mcb_long
#define atm_elen  un_mcb.mcbits.mcb_elen
#define atm_crc   un_mcb.mcbits.mcb_crc
#define atm_loss  un_mcb.mcbits.mcb_loss
#define atm_delay un_mcb.mcbits.mcb_delay
#define atm_pid   un_mcb.mcbits.mcb_pid

```

48
atm.h
-10-

```
#define ATM_PID_LLC 1 /* protocol ID for LLC */
#define ATM_MCBITS_NOCRC 0x04000000 /* protocol id 1 */
#define ATM_HDR_LEN sizeof(struct atm_header)
#define ATM_PAD_SHIFT 24

/*
 * The only header extension defined is a return port address. The
 * length must be set to ATME_RPA_SIZE. Pad exists to get the 64 bit
 * address 64 bit aligned relative to the atm header.
 */

struct atm_header_ext {
    u_char    atme_len;
    u_char    atme_type;
    u_char    atme_pad[2]; /* need not be zeros (nnbz) */
    struct atm_addr atme_rpa; /* return port address */
};

#define ATME_RPA_TYPE 112 /* out of SMDS range */
#define ATME_RPA_BYTES sizeof(struct atm_header_ext)
#define ATME_RPA_WORDS ((sizeof(struct atm_header_ext)+3)/4)

/*
 * Callers to atm_data_req() must ensure atleast ATM_DATA_REQ_ROOM
 * bytes are available in front of the packet data.
 */
#define ATM_DATA_REQ_ROOM (ATM_HDR_LEN+LLC_SNAP_LEN+ATME_RPA_BYTES)

/*
 * multicast address structures are linked to atm_arptabs which are
 * marked ATF_MULTI. Such entries are not timed out, nor are they
 * freed when underlying VCs are released. atm_delete_lan() free's
 * the ATF_MULTI atm_arptab entries and atm_add_lan() &
 * atm_niu_to_niu() re-allocate them and re-initiate MC VCs for the
 * registered addresses.
 */
struct mcaddr {
    u_char    mc_enaddr[6]; /* multicast address */
    u_short   mc_count; /* reference count */
    struct aate *mc_at; /* multicast VC */
};

#define MCADDRMAX 64 /* multicast addr table length */
#define MCCOUNTMAX (32*1024-1) /* multicast addr max
 * reference count */

/*
 * atmif, one per atm lan, used by atm lan layer
 */
struct atmif {
```

49
atm.h
-11-

```

struct niu_arpcorn *ati_ac; /* contains arp and ifnet
    * structures */
struct atmif *ati_next; /* linked off pcif structure */
u_short      ati_state; /* basically do we know who
    * we are */
u_short      ati_mid; /* mid used for multicast frames */
u_short      ati_mcasts; /* max # multicasts circuits
    * configured */

struct atm_addr ati_port;
struct atm_addr ati_mac;
#define ac_mac      ati_mac.aa_byte[2]

struct pcif *ati_pcif;
struct aate *ati_arptab; /* set at initialization */
int      ati_num_mcasts;
struct mcaddr ati_mcaddrs[MCADDRMAX];
};

/* ati_state */
#define ATS_INACTIVE 0
#define ATS_ACTIVE 3

/*
 * global data structure for r/w variables and variables explicitly
 * initialized.
 */

#include "llc.h"

struct atm_globs {
    struct if_tr_hdr *lrb;
    struct atmif *atmif;
    int      atmifn;
    int      atmif_used;
    struct llc_snap llc_def;
    struct atm_addr atm_broadcast;
    struct atm_addr atm_null;
    struct ulptab *atm_ulp;
    int      atm_initialized;
    char      static_buf[32];
};

#ifdef RT68K
extern struct atm_globs atm_globs;
#define atm_glob (&atm_globs)
#else
#define atm_glob atm_get_glob()
struct atm_globs *atm_get_glob();
#endif

```

50
atm.h
-12-

```
#define LEN_FOR_MBUF_PTRS 0xf5560002

#define HASH_MULTICAST_ADDRESS(x) ((x)&0xff)

caddr_t      atm_alloc_msg(), atm_alloc_bytes();
#define NATMS 4 /* max # of ATM lans per physical
               * interface */

#define e160_ntoa svc_e164_ntoa/* these are really E.164 addresses */

#endif      /* NIU_ATM_H */
```

51
atm_init.c
-13-

```

/* atm_init.c
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 *
 * This file contains the ATM LAN configuration routines. They are
 * called by the ATM signaling module when signaling enters the
 * ACTIVE state and one or more ATM LANs have been provisioned
 * "added" by NM and they are called when signaling transitions to
 * the WGRC state to "delete" the ATM LANs.
 *
 * atm_attach_lan() allocates atmif structures and initializes them.
 * atm_add_lan() activates an ATM LAN. atm_delete_lan()
 * deactivates an ATM LAN. atm_niu_to_niu() activates an ATM LAN in
 * back to back configuration. atm_trace_buf() add a trace record to
 * the trace buffer. atm_trace_str() add a string to the trace
 * buffer.
 */

static char sccsid[] = "%A%";

#include "debug.h"
#include "niu.h"

#include "atm.h"
#include "atmarp.h"
#include "svc.h"
#include "debug.h"
#include "trace.h"
#include "if_atm.h"

#define TL1 1
#define TL2 atm_trace>1
#define TL3 atm_trace>2
#define TL4 atm_trace>3
#define TL5 atm_trace>4

extern int atm_trace;

/*
 * atm_attach_lan() is called when the maximum number of ATM LANs for
 * a particular physical interface is known. The appropriate number
 * of atm_lan interface structures are allocated and linked into the
 * physical interface structure.
 */
atm_attach_lan(atp, pc)
    struct atmif *atp;
    struct pcif *pc;
{
    struct atmif *an;

```

52
atm_init.c
-14-

```

if (an = pc->pc_atmif) {
    while (an->ati_next)
        an = an->ati_next;
    an->ati_next = atp;
    atp->ati_next = 0;
} else {
    atp->ati_next = 0;
    pc->pc_atmif = atp;
}
atp->ati_pcif = pc;

ASSERT(atp->ati_state == ATS_INACTIVE);
if_set_mac(atp);
atm_arptab_alloc(atp);
atm_bzero(atp->ati_mcaddrs, sizeof(atp->ati_mcaddrs));
atp->ati_num_mcasts = 1;
atp->ati_mcaddrs[0].mc_count = 1;
atm_bcopy(&atm_glob->atm_broadcast_aa_byte[ATM_FIRST_MAC],
    atp->ati_mcaddrs[0].mc_enaddr, 6);
if (pc->pc_flags & PCIF_NIU_TO_NIU)
    atm_niu_to_niu(atp);
}

/*
 * This is called when a real switch tells use some real port
 * addresses. The mte's were freed when the previous lan was
 * deleted. If the mtu is zero this is a null LAN. It should not be
 * made active. This allows users to configure interfaces starting
 * at aa2, aa3, etc.
 */
atm_add_lan(atp, port, mid, mcasts, mtu)
    struct atmif *atp;
    struct atm_addr *port;
{
    TR1(TL1, "atm add lan: port = %s",
        e160_ntoa(&atp->ati_port));
    if (mtu == 0)
        return;
    atp->ati_state = ATS_ACTIVE;
    atp->ati_mid = mid;
    atp->ati_mcasts = mcasts;
    atp->ati_port = *port;
    if_add_lan(atp, mtu);
    atm_setup_mcasts(atp);
    TR1(TL2, "atm add lan: port = %s",
        e160_ntoa(&atp->ati_port));
    TR1(TL2, " mac = %s\n", e160_ntoa(&atp->ati_mac));
}

```

53
atm_init.c
-15-

```

atm_setup_mcasts(atp)
    struct atmif *atp;
{
    int i;
    struct mcaddr *mc;

    for (i = 0; i < atp->ati_num_mcasts; i++) {
        mc = &atp->ati_mcaddrs[i];
        ASSERT(mc->mc_at == 0);
        mc->mc_at = atm_find_at(atp, mc->mc_enaddr);
        ASSERT(mc->mc_at);
        mc->mc_at->aate_flags |= ATF_MULTI;
        TR1(TL3, "atm_setup_mcast: port = %s\n",
            e160_ntoa(&mc->mc_at->aate_atmaddr));
    }
}

/*
 * Called to free up any resources tied up by the ATM LAN, atp. The
 * arptab has been cleared by release indications except for
 * ATF_MULTI entries. Here we go through list of registered
 * multicast addresses free those arptab entries referenced.
 */
atm_delete_lan(atp)
    struct atmif *atp;
{
    struct mcaddr *mc;

    atp->ati_state = ATS_INACTIVE;
    TR1(TL1, "atm_delete_lan: port = %s",
        e160_ntoa(&atp->ati_port));
    TR1(TL1, " mac = %s\n", e160_ntoa(&atp->ati_mac));
    atp->ati_port_aa_type = AAT_NULL;
    for (mc = atp->ati_mcaddrs;
        mc < &atp->ati_mcaddrs[atp->ati_num_mcasts]; mc++) {
        ASSERT(mc->mc_count);
        if (mc->mc_at) {
            ASSERT(mc->mc_at->aate_flags & ATF_MULTI);
            mc->mc_at->aate_flags = 0;
            atm_aate_free(mc->mc_at);
            mc->mc_at = 0;
        }
    }
    if_delete_lan(atp);
}

/*
 * set a local port address and mac address based upon mac address in
 * niu_arpcom referenced by atp. This is used when changing atm lan
 * configuration to a niu-to-niu configuration. Also set to

```

54
atm_init.c
-16-

```

* broadcast mte entry and if signaling not debugged install a nalled
* up broadcast vc.
*/

```

```

atm_nlu_to_nlu(atp)
struct atmif *atp;
{
    extern int    gosig, niu_mtu;

    atm_bzero(&atp->ati_port, sizeof(atp->ati_port));
    atp->ati_port.aa_type = AAT_PORT;
    atm_bcopy(&atp->ati_mac.aa_byte[ATM_FIRST_MAC],
              &atp->ati_port.aa_byte[ATM_N2N_MAC], 6);
    atp->ati_port.aa_lanum = if_get_lan(atp);
    atp->ati_state = ATS_ACTIVE;
    atp->ati_mcasts = 32; /* its arbitrary */
    atp->ati_mid = 0; /* must be zero till ALAN aal driver
                      * gets fixed */
    if_add_lan(atp, 0); /* do not change mtu */
    atm_setup_mcasts(atp);
    TR1(TL2, "atm_nlu_to_nlu: port = %s",
        e160_ntoa(&atp->ati_port));
    TR1(TL2, " mac = %s\n", e160_ntoa(&atp->ati_mac));
}

svc_trace_pdu(p, len, in, vci)
char *p;
{
    atm_trace_buf(p, atm_bcopy, SVC_PDU_TRACE, len, in, vci);
}

/*
* atm_trace_buf() add a trace record to the trace buffer.
*/
int    atm_trace_limit = 64;

atm_trace_buf(p, copyproc, sub, tlen, in, vci)
caddr_t p;
int    (*copyproc) ();
{
    struct if_tr_hdr *rb;
    int    s, len;

    if (tlen > atm_trace_limit)
        len = atm_trace_limit;
    else
        len = tlen;
    rb = (struct if_tr_hdr *) tr_get_entry(sizeof *rb + len);
    atm_glob->ltrb = rb;
    if (!rb)

```

55
atm_init.c

-17-

```
    return;
    rb->thdr.subsystem = sub;
    rb->thdr.sss = in;
    rb->thdr.length = len;
    rb->tlen = tlen;
    atm_settime(rb->thdr.time);
    rb->vcl = vcl;
    (*copyproc) (p, (char *) &rb[1], len);
}

/*
 * atm_trace_str() add a string to the trace buffer.
 */
atm_trace_str(str)
char      *str;
{
    struct trace_header *tr;
    int      len;

    for (len = 0; str[len]; len++);
    tr = (struct trace_header *) tr_get_entry(sizeof *tr + len);
    if (!tr)
        return;
    tr->subsystem = ASCII_LOG;
    tr->sss = 0;
    tr->length = len;
    atm_settime(tr->time);
    atm_bcopy((u_char *) str, (u_char *) &tr[1], len);
}
```

56
atmarp.c
-18-

```

/* atmarp.c
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 *
 * ATM address resolution protocol. This module contains the routines
 * which implement atm arp. The two primary entry points are
 * atm_mac_to_aate() and atm_arp_input().
 *
 * atm_mac_to_aate() returns a pointer to an atm arp table entry.
 * atm_arp_input() handles all atm arp requests and replies.
 */

static char    sccsid[] = "%A%";

#ifdef notdef
#include "all.h"
#include "ip_errs.h"
#include "unsp.h"
#endif /* notdef */

#include "atm.h"
#include "svc.h"
#include "atmarp.h"
#include "debug.h"
#include "if_atm.h"

#define TL1      1
#define TL2      arp_trace > 1
#define TL3      arp_trace > 2
#define TL4      arp_trace > 3
#define TL5      arp_trace > 4
int             arp_trace = 2;
int             arp_debug = 2;
char            *atm_mac_printf();
#define N_AATE_S 64
struct aate     atm_aate[N_AATE_S * ATM_ARP_TABLES];
int             atm_aate_size = N_AATE_S * ATM_ARP_TABLES;
int             atm_arptabs = 0; /* number of tables allocated (1 per
 * atm lan) */

struct aate     *
atm_arptab_look(atp, addr)
    struct atmif *atp;
    u_char      *addr;
{
    struct aate  *ate = atp->ati_arptab, *end;

```

57
atmarp.c
-19-

```

    end = &ate[N_AATE_S];
    while (ate < end) {
        if (atm_bcmp(ate->aate_macaddr, addr,
                    sizeof(ate->aate_macaddr)) == 0)
            return ate;
        else
            ate++;
    }
    return 0;
}

/*
 * Broadcast an ATM_ARP packet, asking who has addr on atm lan ac.
 */
atm_arprequest(atp, addr)
    struct atmif *atp;
    u_char *addr;
{
    struct atm_arp *aa;

    TR2(TL3, "atm_arprequest(%x, %s)\n", atp,
        atm_mac_sprintf(addr, 6));
    aa = (struct atm_arp *) atm_alloc_msg0;
    if (!aa)
        return;
    aa->aa_llp = htons(ARPHRD_ATM);
    aa->aa_ulp = htons(ETHERTYPE_ATMMAC);
    aa->aa_llp_len = sizeof(aa->aa_sender_port);
    aa->aa_ulp_len = sizeof(aa->aa_sender_mac);
    aa->aa_msg_type = htons(ATM_ARP_REQUEST);
    atm_bcopy((caddr_t) &atp->atf_port, (caddr_t) aa->aa_sender_port,
              (u_int) sizeof(aa->aa_sender_port));
    atm_bcopy((caddr_t) &atp->ac_mac, (caddr_t) aa->aa_sender_mac,
              (u_int) sizeof(aa->aa_sender_mac));
    atm_bcopy((caddr_t) addr, (caddr_t) aa->aa_target_mac,
              (u_int) sizeof(aa->aa_target_mac));
    atm_bzero((caddr_t) aa->aa_target_port, sizeof(aa->aa_target_port));
    atm_send_arp(atp, addr, aa, sizeof(*aa));
}

/*
 * atm_mac_to_aate() returns a pointer to and atm arp table entry.
 *
 * desten is filled in. If there is no entry in arptab, set one up and
 * broadcast a request for the MAC address. An arptab point is
 * returned if an existing arptab entry was found. for the mac
 * address is found (or computed).
 */

```

58
atmarp.c
-20-

```

struct aate *
atm_mac_to_aate(ac, destmac)
    struct atmif *ac;
    u_char *destmac;
{
    struct aate *at;

    TR2(TL4, "atm_mac to aate(%x, %s)\n", ac,
        atm_mac_sprintf(destmac, 6));
    at = atm_arptab_lookup(ac, destmac);
    if (at == 0) { /* not found */
        at = atm_aate_alloc(ac->atm_arptab, destmac);
        if (at == 0)
            panic("atm_mac to aate: no free entry");
        if (!atm_bcmp(destmac, &ac->ac_mac, 6)) {
            atm_bcopy((caddr_t) &ac->atm_port,
                (caddr_t) &at->aate_atmaddr,
                (u_int) sizeof(ac->atm_port));
            atm_bcopy((caddr_t) &ac->ac_mac,
                (caddr_t) at->aate_macaddr,
                (u_int) sizeof(at->aate_macaddr));
            at->aate_timer = 0;
            at->aate_flags = AATF_COMPLETE;
            at->aate_vcte = 0;
            TR2(TL4, " -> %x %s\n", at,
                svc_e164_ntoa(&at->aate_atmaddr));
            return at;
        } else if (destmac[0] & 0x1) {
            /*
             * Calculate a suitable atm address to make a
             * connection with for this multicast
             * address.
             */
            atmarpmhash(ac, destmac,
                (caddr_t) &at->aate_atmaddr);
            atm_bcopy(destmac, (caddr_t) at->aate_macaddr,
                (u_int) sizeof(at->aate_macaddr));
            at->aate_timer = 0;
            at->aate_flags = AATF_COMPLETE;
            at->aate_vcte = 0; /* no vcte get */
            TR2(TL4, " -> %x %s\n", at,
                svc_e164_ntoa(&at->aate_atmaddr));
            return at;
        }
    }
    /*
     * Generate an ARP request, AATF_RESOLVING avoids
     * recursion.
     */
    at->aate_flags |= AATF_RESOLVING;
    atm_arprequest(ac, destmac);
}

```

59
atmarp.c
-21-

```

    at->aate_flags &= ~AATF_RESOLVING;
    TR0(TL4, "-> 0\n");
    return 0;
}
at->aate_timer = 0; /* restart the timer */
if (at->aate_flags & AATF_COMPLETE) { /* entry IS complete */
    TR2(TL4, "-> %x %s\n", at,
        svc_e164_ntoa(&at->aate_atmaddr));
    return at;
}
/*
 * There is an aate entry, but no e164 address response yet.
 * Avoid infinite recursion by using the AATF_RESOLVING flag.
 * This is temporary. The port to rt68k will necessitate
 * restructuring which, hopefully, will remove the need for
 * this flag.
 */
if (!(at->aate_flags & AATF_RESOLVING)) {
    at->aate_flags |= AATF_RESOLVING;
    atm_arprequest(ac, destmac);
    at->aate_flags &= ~AATF_RESOLVING;
}
TR0(TL4, "-> 0\n");
return 0;
}

/*
 * atm_arp_input() handles all atm arp requests and replies.
 */
atm_arp_input(atp, aa, src)
    struct atmif *atp;
    struct atm_arp *aa;
    u_char *src;
{
    struct aate *at;
    u_char target_mac[6]; /* copy of target protocol
        * address */
    struct atm_addr *sport;

    TR2(TL2, "atm_arp_input(from %s / %s)\n",
        atm_mac_sprintf(aa->aa_sender_mac, 6), svc_e164_ntoa(aa->aa_sender_port));

    if (aa->aa_ulp != ETHERTYPE_ATMMAC)
        goto drop;

    /* make a copy of target mac for use later */
    atm_bcopy((caddr_t) aa->aa_target_mac, (caddr_t) target_mac,
        (u_int) sizeof(target_mac));

    if (!atm_bcmp((caddr_t) aa->aa_sender_mac, (caddr_t) &atp->ac_mac,

```

60
atmarp.c

-22-

```

        sizeof(aa->aa_sender_mac)))
/* its from me so just drop it */
goto drop;
/*
 * Search the local database for senders mac address. Update
 * the database with new information (first deleting old
 * information).
 */
at = atm_arptab_look(atp, aa->aa_sender_mac);
if (at) { /* if at is complete and address is
 * different then free entry */
    if (at->aate_flags & AATF_COMPLETE) {
        if (atm_bcmp(aa->aa_sender_port, (caddr_t) & at->aate_atmaddr,
            (u_int) sizeof(aa->aa_sender_port)) != 0) { /* if different */
            atm_aate_free(at);
            at = 0;
        }
    } else { /* if at is incomplete update it */
        atm_bcopy((caddr_t) aa->aa_sender_port, (caddr_t) & at->aate_atmaddr,
            sizeof(aa->aa_sender_port));
        at->aate_flags |= AATF_COMPLETE;
    }
}
/*
 * If we didn't find his mac address and he IS looking for
 * us. Then learn his as well
 */
if (at == 0 && !atm_bcmp(target_mac, &atp->ac_mac, sizeof(target_mac))) {
    /* ensure we have a table entry */
    if (at = atm_aate_alloc(atp->ati_arptab, aa->aa_sender_mac)) {
        atm_bcopy((caddr_t) aa->aa_sender_port, (caddr_t) & at->aate_atmaddr,
            (u_int) sizeof(aa->aa_sender_port));
        at->aate_flags |= AATF_COMPLETE;
    }
}
/*
 * If we found his mac address and his mac address is NOT the
 * same as the guy we got this frame from, then set
 * AATF_PROXY.
 */
if (at && atm_bcmp((caddr_t) src, (caddr_t) aa->aa_sender_mac,
    (u_int) sizeof aa->aa_sender_mac))
    at->aate_flags |= AATF_PROXY;
reply:
/*
 * Make sure that you are trying to resolve a MAC address vs
 * some other type of address.
 */
if (aa->aa_msg_type != ATM_ARP_REQUEST)
    goto drop;

```

61
atmarp.c
-23-

```

if (!atm_bcmp(target_mac, &atp->ac_mac, (u_int) sizeof(target_mac)) ||
    atm_mac_learned_on_non_atm_interface(target_mac))
/*
 * Either we are the target of the arp request or we
 * found the target mac address is in our forwarding
 * database and it was learned on a non-ATM interface
 * so we send a proxy atm arp response because the
 * atm arp request can not be forwarded onto that
 * link.
 */
    sport = &atp->atl_port;
else
    goto drop;

/*
 * We have decided to respond. Turn the request into a
 * response by copying the sender's port address into the
 * target port adr. Copy the senders protocol (MAC) address
 * into the target address. Copy the target protocol address
 * (target_mac) into the senders protocol address and copy
 * the port selected above into the senders port address.
 */

atm_bcopy((caddr_t) aa->aa_sender_port, (caddr_t) aa->aa_target_port,
           (u_int) sizeof(aa->aa_sender_port));
atm_bcopy((caddr_t) aa->aa_sender_mac, (caddr_t) aa->aa_target_mac,
           (u_int) sizeof(aa->aa_sender_mac));
atm_bcopy((caddr_t) target_mac, (caddr_t) aa->aa_sender_mac,
           (u_int) sizeof(aa->aa_sender_mac));
atm_bcopy((caddr_t) sport, (caddr_t) aa->aa_sender_port,
           (u_int) sizeof(aa->aa_sender_port));
aa->aa_msg_type = htons(ATM_ARP_REPLY);

/* go ahead and send it */
atm_send_arp(atp, (caddr_t) aa->aa_target_mac, aa, sizeof(*aa));
return;
drop:
    atm_free_msg((char *) aa);
    return;
}

/*
 * atm_arptab_alloc() allocates an atm arp table for the atmif
 * structure atp and schedules the first first timeout for that atm
 * arp table. Each atm arp table has uts timeouts scheduled
 * separately. This is an attempt to spread signaling traffic out
 * when there are multiple LANs. Zero is returned if no table could
 * be allocated.
 */

```

62
atmarp.c
-24-

```

struct aate *
atm_arptab_alloc(atp)
{
    struct atmif *atp;
    struct aate *at;

    at = &atm_aate[N_AATE_S * atm_arptabs];
    if (atm_arptabs == ATM_ARP_TABLES)
        return 0;
    atp->ati_arptab = at;
    atm_sched_timeout(atp);
    atm_arptabs++;
    return at;
}

/*
 * atm_aate_free() is called when an atm arp table entry is no longer
 * in use. If a VC is referenced its reference count is decremented.
 * If the VC's reference goes to zero atm_release() is called to
 * release the VC.
 */

atm_aate_free(at)
{
    struct aate *at;
    {
        TR2(TL3, "atm_aate free(%s / %s)\n", atm_mac_sprintf(at->aate_macaddr, 6),
            svc_e164_ntoa(&at->aate_atmaddr));
        ASSERT((at->aate_flags & AATF_MULTICAST) == 0);
        at->aate_timer = at->aate_flags = 0;
        if (at->aate_vcte && (svc_dec(at->aate_vcte) == 0))
            atm_release(at->aate_vcte, VC_IDLE);
        at->aate_vcte = 0;
    }
}

/*
 * Enter a new address in aate, pushing out the oldest entry from the
 * bucket if there is no room. This always succeeds since no bucket
 * can be completely filled with permanent entries (except from
 * atm_arpioctl when testing whether another permanent entry will
 * fit).
 */

struct aate *
atm_aate_alloc(at, addr)
{
    struct aate *at; /* base of aate table for this lan */
    u_char *addr;
    {
        int hiwater = 0;
        struct aate *victim = 0, *end = &at[N_AATE_S];
    }
}

```

63

atmarp.c
-25-

```

TR1(TL3, "atm_aate_alloc(%s)\n", atm_mac_sprintf(addr, 6));
while (at < end) {
    if (at->aate_flags == 0)
        goto found;
    if ((at->aate_flags & AATF_MULTICAST) == 0
        && (at->aate_timer >= hiwater)) {
        hiwater = at->aate_timer;
        victum = at;
    }
    at++;
}
if (!victum)
    return 0;
at = victum;
atm_aate_free(at);
found:
atm_bcopy((caddr_t) addr, (caddr_t) at->aate_macaddr,
          (u_int) sizeof(at->aate_macaddr));
at->aate_flags = AATF_INUSE;
at->aate_vcte = 0;
return (at);
}

/*
 * atmarphash() - algorithmically generates an ATM multicast address
 * from a 48 bit address. If the number of multicast circuits
 * supported on this LAN (ati_mcasts) is 0 then use all 48 bits of
 * the MAC multicast address. The assumption being the network is
 * providing multicast service via a server with no limit to number
 * of multicast connections available to each station. If ati_mcasts
 * is greater than zero then the 48 bit address is folded into a
 * unsigned 32 bit integer by exclusive or'ing the first 2 bytes into
 * the last two bytes. The resulting integer is divided by ati_mcasts
 * and the resulting number is used as the ATM multicast address.
 */

atmarpmhash(atp, mac, port)
struct atmif *atp;
u_char *mac;
struct atm_addr *port;
{
    port->aa_long[0] = 0; /* clear first word of address */
    if (atp->ati_mcasts == 0) {
        atm_bcopy(mac, &port->aa_byte[ATM_FIRST_MAC], 6);
    } else {
        port->aa_long[1] = ((mac[2] << 24) |
                           ((mac[3] ^ mac[1]) << 16) |
                           ((mac[0] ^ mac[4]) << 8) |
                           (mac[5]));
        port->aa_long[1] %= atp->ati_mcasts;
    }
}

```

64
atmarp.c

-26-

```

/*
 * This must be flipped in order to get it into
 * network byte order
 */
htonl(port->aa_long[1]);
port->aa_byte[ATM_FIRST_MAC] = 1; /* set multicast bit */
}
TR1(TL4, "computed mcast of %x\n", port->aa_long[1]);
port->aa_type = AAT_MAC; /* set the type field */
}

/*
 * arp_setup() is called when a new VC is setup. The arptable is
 * searched for entries needing a VC to the peer port. Those entries
 * are updated to reference the new VC.
 */

arp_setup(at, vp)
    struct aate *at;
    struct vcte *vp;
{
    int i;
    TR1(TL2, "atm_setup(%s)\n", svc_e164_ntoa(&vp->vcte_peer));

    for (i = 0; i < N_AATE_S; i++, at++) {
        if (!(at->aate_flags & AATF_COMPLETE))
            continue;
        if (!atm_bcmp(&at->aate_atmaddr, &vp->vcte_peer,
                     sizeof(vp->vcte_peer)) &&
            at->aate_vcte != vp) {
            at->aate_vcte = vp;
            svc_inc(vp);
        }
    }
}

/*
 * arp_release() is called when a VC is about to be released. The
 * arptable is searched for references to the VC. Any entries found
 * are freed.
 */

arp_release(at, vp)
    struct aate *at;
    struct vcte *vp;
{
    int i;
    TR1(TL2, "arp_release(%s)\n", svc_e164_ntoa(&vp->vcte_peer));
    for (i = 0; i < N_AATE_S; i++, at++) {
        if (!(at->aate_flags & AATF_COMPLETE)) {

```

65
atmarp.c
-27-

```

    ASSERT(at->aate_vcte == 0);
    continue;
}
if (at->aate_vcte != vp)
    continue;
at->aate_vcte = 0;
svc_dec(vp);
if ((at->aate_flags & AATF_MULTICAST) == 0)
    atm_aate_free(at); /* must clear aate_vcte
                        * first */
else
    TR1(TL2, "arp_release: skipped AATF_MULTICAST at=%x\n", at);
}
ASSERT(vp->vcte_refcnt == 0);
}

int      atmarp_timeo1 = 24; /* idle timeout */
int      atmarp_timeo2 = 3; /* incomplete timeout */
extern int hz;
/*
 * atm_arptimer() scans arp tables for active the ATM LAN atp. VCs ar
 * established fo registered multicast addresses if none exist. Atm
 * arp entries for none registered addresses are timed out. When the
 * last reference to a VC is freed that VC is released (atm_aate_free
 * does this).
 */

atm_arptimer(atp)
struct atmif *atp;
{
    struct aate *at = atp->atl_arptab, *end;
    int s;

    TR1(TL3, "atm_arptimer(%x)\n", at);
    atm_sched_timeout(atp);
    if (atp->ati_state == ATS_INACTIVE)
        return;
    s = splimp();
    end = &at[N_AATE_S];
    for (; at < end; at++) {
        /* set up multicast circuits which have been released */
        if ((at->aate_flags & AATF_MULTICAST) &&
            (at->aate_vcte == 0 ||
             ((1 << at->aate_vcte->vcte_state) & VCS_DEAD_OR_DYING))) {
            atm_initiate_setup(atp, at);
            continue;
        }
        if (at->aate_flags == 0 ||
            (at->aate_flags & AATF_MULTICAST))
    }
}

```

66
atmarp.c
-28-

```
        continue;
    at->aate_timer++;
    if (at->aate_flags & AATF_COMPLETE) {
        if (at->aate_timer >= atmarp_timeo1)
            atm_aate_free(at);
        } else if (at->aate_timer >= atmarp_timeo2)
            atm_aate_free(at);
    }
    spbx(s);
}
```

67
atmarp.h
-29-

```

/* atmarp.h
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED

#ifndef RT68K
#include "sys/types.h"
#else
#include <stdint.h>
#endif

/*
 * ATM Address Resolution Protocol (or ATM ARP) Definitions.
 */

#define ETHERTYPE_ATMMAC 0x0805
#define ARPHRD_ATM 16
/*
 * ATM Address Resolution Protocol.
 */
struct atm_arp {
    u_short    aa_llp; /* lower layer protocol */
    u_short    aa_ulp; /* upper layer protocol */
    u_char     aa_llp_len;
    u_char     aa_ulp_len;
    u_short    aa_msg_type;
    u_char     aa_sender_port[8];
    u_char     aa_sender_mac[6];
    u_char     aa_target_port[8];
    u_char     aa_target_mac[6];
};
/* aa_msg_type's */
#define ATM_ARP_REQUEST 1
#define ATM_ARP_REPLY 2
/*
 * MAC to ATM address resolution table, atm arp table entry, aate.
 */
struct aate {
    struct atm_addr aate_atmaddr; /* port address */
    struct vcte *aate_vcte; /* vcte reference */
    u_char     aate_macaddr[6]; /* mac address */
    u_char     aate_timer; /* ticks */
    u_char     aate_flags; /* flags */
};
/* aate_flags field values */
#define AATF_INUSE 0x01
#define AATF_COMPLETE 0x02
#define AATF_MULTICAST 0x10
#define ATF_MULTI AATF_MULTICAST
#define AATF_PROXY 0x20

```

68
atmarp.h
-30-

```
#define AATF_RESOLVING 0x40
```

```
struct aate *atm_aate_alloc(), *atm_arptab_look(), *atm_arpresolve();  
struct aate *atm_arptab_alloc(), *atm_find_at();  
extern struct aate atm_aate[];  
#define ATM_ARP_TABLES 16  
extern int atm_arptabs;
```

69
bits.c
-31-

```
/* bits.c
```

```
*
```

```
* COPYRIGHT 1992 ADAPTIVE CORPORATION
```

```
* ALL RIGHTS RESERVED
```

```
*/
```

```
*****END*****/
```

```
#ifndef UNIX
```

```
#define ERRLOG printdbg
```

```
#define printf printdbg
```

```
#endif /* ifndef UNIX */
```

```
#include <stdint.h>
```

```
#include "bits.h"
```

```
bits_get_bit(bits, size)
```

```
bits_t *bits;  
int size;
```

```
{  
    int max_bit;  
    int ret;  
    int i;  
    bits_t mask;
```

```
    max_bit = size * 8 * SIZE_BITS;  
    for (ret = 0; ret < max_bit; ret++) {  
        BITS_GET_1_MASK(ret, i, mask);  
        if ((bits[i] & mask) == 0) {  
            bits[i] |= mask;  
            return (ret);  
        }  
    }
```

```
    return (-1);  
}
```

```
bits_tst_bit(bit, bits, size)
```

```
int bit;  
bits_t *bits;  
int size;
```

```
{  
    int ret;  
    int i;  
    bits_t mask;
```

```
    BITS_GET_1_MASK(bit, i, mask);  
    ret = i < size && (bits[i] & mask) != 0;  
    return (ret);  
}
```

70
bits.c
-32-

```

bits_alloc_bit(bit, bits, size)
    int      bit;
    bits_t   *bits;
    int      size;
{
    int      ret;
    int      i;
    bits_t   mask;

    if (!bits || !bit || (bit >= size)) {
        BITS_GET_I_MASK(bit, i, mask);
        if (i < size) {
            bits[i] |= mask;
            return (-1);
        } else {
            return (0);
        }
    }
    return (0);
}

bits_free_bit(bit, bits, size)
    int      bit;
    bits_t   *bits;
    int      size;
{
    int      i;
    bits_t   mask;

    BITS_GET_I_MASK(bit, i, mask);
    if (i < size)
        bits[i] &= ~mask;
}

print_bits(bits, size)
    bits_t   *bits;
    int      size;
{
    int      i;

    for (i = 0; i < size; i++) {
        if (bits[i] == 0) {
            printf("0x0 ");
        } else {
            printf("0x%08x ", bits[i]);
        }
    }
}

```

//
bits.h
-33-

/* bits.h

*

* COPYRIGHT 1992 ADAPTIVE CORPORATION

* ALL RIGHTS RESERVED

*/

*****END*****/

#ifndef BITS_H

#define BITS_H

typedef tUINT32 bits_t;

#define SIZE_BITS (sizeof(bits_t))

#define BITS_GET I MASK(bit, i, mask) \

((i) = (bit) / (8 * SIZE_BITS), \

(mask) = (bits_t)0x80000000 >> ((bit) % (8 * SIZE_BITS)))

#endif

/* ifdef BITS_H */

72

if_atm.c
-34-

```
/* if_atm.c
*
* COPYRIGHT 1992 ADAPTIVE CORPORATION
* ALL RIGHTS RESERVED
*
*
* This file contains the operating specific routine for the ATM LAN MAC
* for UNIX. The routines defined here are: atm_attach() is called
* once per phys i/f at initialization.
*
* if_set_mac() is called get a MAC address from the OS.
*
* if_add_lan() is notify the OS an ATM LAN is ACTIVE.
*
* if_delete_lan() is notify the OS an ATM LAN is INACTIVE.
*
* atm_free_packets() is called to free packets queued on a VC.
*
* atm_append_packet() appends a packet on VC til it comes up.
*
* atm_send_packets() is called when a VC becomes established.
*
* atm_settime() Copy OS notion of time into long array of 2.
*
* atm_sched_timeout() schedule the OS to call atm_arptimer().
*
* svc_sched_timeout() schedule the OS to call svc_timeout().
*
* svc_init() allocates global memory area for ATM signaling.
*
* svc_report_version_conflict() report signaling version conflicts to
* OS.
*
* atm_alloc_msg() allocates a msg buffer large enough for signal PDUs.
*
* atm_free_msg() frees message memory pointed to by cp.
*
* atm_alloc_bytes() is used to allocate memory for data structures
*
* aal_send_msg() sends the psecified aal payload on a VC.
*
* atm_send_arp() send a frame encapsulating in 802.6/LLC/SNAP type=ARP.
*
* atm_mac_input() handles aal payloads with 802.6 PDUs (ATM LAN).
*
* svc_mac_input() handles aal payloads with SVC PDUs.
*/
```

73

if_atm.c
-35-

```

static char    sccsid[] = "%A%";

#ifdef INET
#define INET    /* only support Internet addressing */
#endif
#include "../sys/param.h"
#include "../sys/systm.h"
#include "../sys/mbuf.h"
#include "../sys/socket.h"
#include "../sys/errno.h"
#include "../sys/lock.h"

#include "../net/if.h"
#include "../net/netisr.h"
#include "../net/route.h"
#include "../net/if_arp.h"

#include "../sun/openprom.h"
#include "../sun4c/mmu.h"

#ifdef INET
#include "../netinet/in.h"
#include "../netinet/in_systm.h"
#include "../netinet/in_var.h"
#include "../netinet/ip.h"
#endif

#include "debug.h"
#include "niu.h"
#include "unipdu.h"
#include "atm.h"
#include "llc.h"
#include "svc.h"
#include "svc_util.h"
#include "if_niuar.h"
#include "atmarp.h"
#include "if_nlu.h"
#include "if_nluo.h"
#include "if_atm.h"
#include "sys/time.h"

int            atm_watch();
int            svc_pcm = 1;
extern int     atm_trace;
#define TL1    1
#define TL2    atm_trace > 1
#define TL3    atm_trace > 2
#define TL4    atm_trace > 3
#define TL5    atm_trace > 4

```

74
if_atm.c
-36-

```

struct niu_arpcom niu_arpcoms[NNIU * NATMS];
/*
 * atm_attach() is called when the maximum number of allowed ATM LANs
 * for a specific interface have been determined. atm_nnius must be
 * patched with the number of ATM LANs per physical interface before
 * booting. Only that number of ATM LANs will be configured
 * regardless of ATM LAN configuration information provided by the
 * LMI configuration protocol. (We could dynamically attach and
 * detach ifnet structures BUT there is a significant
 * probability other code in the system assumes ifnets are static
 * after boot. So ifnets are statically linked once at boot.)
 */

atm_attach(pc)
    struct pcif *pc;
{
    struct ifnet *ifp;
    struct atmif *atp;
    int lan, ifunit;
    int niuoutput(), niuoioclt(), mac[2];

    atm_init();
    for (lan = 0; lan < pc->pc_num_lans; lan++) {
        ifunit = atm_glob->atmif_used++;
        atp = &atm_glob->atmif[ifunit];
        atp->ati_ac = &niu_arpcoms[ifunit];
        niu_arpcoms[ifunit].ac_atmif = atp;
        ifp = &atp->ati_ac->ac_if;
        ifp->if_unit = ifunit;
        if (niu_info[pc->pc_num].macaddr[0] ||
            niu_info[pc->pc_num].macaddr[1]) {
            mac[0] = niu_info[pc->pc_num].macaddr[0];
            mac[1] = niu_info[pc->pc_num].macaddr[1] + lan;
            bcopy(((char *) mac) + 2,
                ((struct niu_arpcom *) ifp)->ac_enaddr, 6);
        } else
            niu_get_enaddr(ifp->if_unit,
                ((struct niu_arpcom *) ifp)->ac_enaddr);
        ifp->if_name = "aa";
        ifp->if_mtu = pc->pc_hw_max_mtu;
        ifp->if_flags = IFF_BROADCAST | IFF_NOTRAILERS;
        ifp->if_loctl = niuoioclt;
        ifp->if_output = niuoutput;
        ifp->if_promisc = 1;
        ifp->if_timer = .1;
        ifp->if_snd.ifq_maxlen = IFQ_MAXLEN;
        ifp->if_watchdog = atm_watch;
        if_attach(ifp);
        atm_attach_lan(atp, pc);
    }
}

```

25

if_atm.c
-37-

```

}

int      awi = 0; /* atm watch interval, for those who
                * can not type */
int      awittg[NNIU]; /* atm watch interval ticks to go */

atm_watch(ifunit)
int      ifunit;
{
    struct ifnet *ifp = (struct ifnet *) & niu_arpcoms[ifunit];
    int      iounit = NIU_IFUNIT_TO_IOUNIT(ifunit);
    int      s;

    ifp->if_timer = 1;
    if (ifunit != 0)
        return 0;
    if (awi == 0)
        return 0;
    if (awittg[iounit] <= 0) {
        printf("reseting unit %d from watchdog\n", iounit);
        s = splimp();
        (*niu_info[iounit].reset) (iounit, 1);
        if (niu_info[iounit].type == NIU_TYPE_HW) {
            setup_rxbuf(&niu_info[iounit]);
        }
        splx(s);
        awittg[iounit] = awi;
    } else
        awittg[iounit]--;
    return 0;
}

/*
 * set ati_mac from value provided by os
 */
if_set_mac(atp)
struct atmif *atp;
{
    bzero(&atp->ati_mac, sizeof(struct atm_addr));
    atp->ati_mac.aa_type = AAT_MAC;
    bcopy(atp->ati_ac->ac_enaddr,
          &atp->ati_mac.aa_byte[ATM_FIRST_MAC], 6);
}

/*
 * if_add_lan() is notify the OS an ATM LAN is ACTIVE. (and
 * conditionally set mtu size)
 */
if_add_lan(atp, mtu)
struct atmif *atp;

```

76
if_atm.c
-38-

```

    int      mtu;
    {
        if (mtu && mtu <= atp->ati_pcif->pc_hw_max_mtu)
            ((struct ifnet *) atp->ati_ac)->if_mtu = mtu;
        ((struct ifnet *) atp->ati_ac)->if_flags |= IFF_RUNNING;
    }

    /*
     * if_delete_lan() is notify the OS an ATM LAN is INACTIVE.
     */

    if_delete_lan(atp, mtu)
        struct atmif *atp;
    {
        ((struct ifnet *) atp->ati_ac)->if_flags &= ~IFF_RUNNING;
    }

    /*
     * convert os ifunit to atmif index, this is not called very often.
     */
    if_get_lan(atp)
        struct atmif *atp;
    {
        int      i;
        struct atmif *atp0;
        for (i = 0, atp0 = atp->ati_pcif->pc_atmif;
            atp0 != atp;
            atp0 = atp0->ati_next, i++);
        return i;
    }

    /*
     * atm_free_packets() is called to free packets queued on a VC.
     */
    #ifndef KERNEL
    #define m_freem(m) atm_free_msg(m)
    #endif

    atm_free_packets(vp)
        struct vcte *vp;
    {
        struct mbuf *m = (struct mbuf *) vp->vcte_packet;
        struct mbuf *m0;

        while (m) {
            m0 = m->m_act;
            m_freem(m);
            m = m0;
        }
        vp->vcte_packet = 0;
    }

```

77
if_atm.c
-39-

```

/*
 * atm_append_packet() appends a packet on VC til it comes up. (we
 * only queue two packets)
 */
atm_append_packet(vp, m)
    struct vcte *vp;
    struct mbuf *m;
{
    m->m_act = 0;
    if (vp->vcte_packet) {
        if (((struct mbuf *) vp->vcte_packet)->m_act != 0) {
            m_freem(m);
            return;
        } else
            ((struct mbuf *) vp->vcte_packet)->m_act = m;
    } else
        vp->vcte_packet = (caddr_t) m;
    svc_glob->svcstat.queued_frames++;
    return;
}
/*
 * atm_send_packets() is called when a VC becomes established. Any
 * queued packets are sent.
 */
atm_send_packets(vp)
    struct vcte *vp;
{
    struct mbuf *m = (struct mbuf *) vp->vcte_packet;
    struct mbuf *m0;

    while (m) {
        m0 = m->m_act;
        aal_send_msg(vp, vp->vcte_atmif->ati_mid, (caddr_t) m,
            LEN_FOR_MBUF_PTRS);
        m = m0;
    }
    vp->vcte_packet = 0;
}
/*
 * atm_settime() Copy OS notion of time into long array of 2.
 */
atm_settime(t)
    struct timeval *t;
{
    extern struct timeval time;
    int spl;
    spl = spl70;

```

78
if_atm.c
-40-

```

    *t = time;
    splx(spl);
}

int      atm_arptimer();
int      atm_arp_ms_per_tick = 10000; /* once every 10 seconds */
/*
 * atm_sched_timeout() schedule the OS to call atm_arptimer().
 */
atm_sched_timeout(atp)
    register struct atmif *atp;
{
#ifdef notdef
    /* I assumed tcp_start_timer() takes seconds ?? */
    tcp_start_timer(&(atp->tmr_entry),
        atm_arp_ms_per_tick / 10 ? ? ?,
        atm_arptimer, (caddr_t) atp);
#else
    timeout(atm_arptimer, (caddr_t) atp,
        atm_arp_ms_per_tick * hz / 1000);
#endif
}

/*
 * returns true if we are forwarding frames out a specific non-ATM
 * interface. Otherwise returns false.
 */

atm_mac_learned_on_non_atm_interface(mac)
    caddr_t mac;
{
    return 0;
}

/*
 * atm_send_arp() send a frame encapsulating in 802.6/LLC/SNAP
 * type=ARP.
 */
atm_send_arp(atp, mac, msg, len)
    struct atmif *atp;
    caddr_t msg, mac;
{
    struct mbuf *m;
    struct sockaddr sa;

    m = dtom(msg);
    m->m_len = len;
    ASSERT(len <= MLEN);
    sa.sa_family = AF_UNSPEC;

```

77
if_atm.c
-41-

```

((struct ether_header *) sa.sa_data)->ether_type =
    ETHERTYPE_ARP;
bcopy(mac,
    ((struct ether_header *) sa.sa_data)->ether_dhost, 6);
niuoutput(atp->atf_ac, m, &sa);
}

/*
 * aal_send_msg() sends the psecified aal payload on a VC. Sends a
 * message, msg, of length, len, byte on VC vp using multiplex-id,
 * mid. If len is LEN_FOR_MBUF_PTR then msg is a pointer to an mbuf
 * chain. Otherwise it is a pointer into an mbuf. There should
 * probably be separate routines for this...
 */
aal_send_msg(vp, mid, msg, len)
    struct vcte *vp;
    int mid;
    caddr_t msg;
    int len;
{
    struct mbuf *m;
    struct ifnet *ifp;
    struct sockaddr_aal sa;

    if (len == LEN_FOR_MBUF_PTRS) {
        m = (struct mbuf *) msg;
    } else {
        struct setup *pdu;
        pdu = (struct setup *) msg;
        TR1(TL3, "aal_send: %s\n",
            svc_pdu_type_str(pdu->lmi_pdu_type));
        if (pdu->lmi_pdu_type <= LMI_PDU_LAST)
            svc_glob->svcstat.pdus_sent[pdu->lmi_pdu_type] ++;
        m = dtom(msg);
        m->m_len = len;
        if ((pdu->lmi_cref_type | pdu->lmi_cref_value) && TL2)
            svc_trace_pdu(pdu, len, 0, vp->vcte_ovpci);
    }
    vcoutput(vp, m, mid);
}

/*
 * atm_mac_input() handles aal payloads processing 802.6 & calling
 * LLC. Should be called at SPLIMP. Called by deliver_packet().
 * atm_data_ind() is called to process the ATM MAC header and get a
 * pointer to the LLC header. llc_data_ind() is called to process
 * the llc frame.
 */

```

86
if_atm.c
-42-

```

atm_mac_input(vp, m0)
    struct mbuf *m0;
    struct vcte *vp;
{
    struct ifnet *ifp;
    struct atm_header *ah;
    struct llc_snap *lp;
    u_char *src;
    u_long *p;
    int mac_hdr_len;
    int promisc = 0;

    ifp = (struct ifnet *) ((struct atmif *) vp->vcte_atmif->ati_ac;
    ifp->if_ipackets++;

    DB1(DL4, "aa%d: atm_mac_input\n", ifp->if_unit);

    ah = mtod(m0, struct atm_header *);
    ASSERT((((u_long) ah) & 0x3) == 0);
    if (m0->m_len < LLC_LEN + ATM_HDR_LEN + ah->atm_elen * 4) {
        DB1(DL1, "aa%d: atm_mac_input short frame\n", ifp->if_unit);
        m_freem(m0);
        return ENOBUFS;
    }
    if (ah->atm_dst.aa_byte[2] & 0x01) {
        if (!niu_findmulti(vp->vcte_atmif, &ah->atm_dst.aa_byte[2])) {
            promisc = 1;
        }
    }
    else if (IATM_ADDR_EQ(ah->atm_dst, vp->vcte_atmif->ati_mac)) {
        promisc = 1;
    }
    lp = (struct llc_snap *) ((caddr_t) ah + ah->atm_elen * 4 +
        sizeof(*ah));
    mac_hdr_len = (caddr_t) lp - (caddr_t) ah;
    src = &ah->atm_src.aa_byte[2];
    if (ifp->if_promisc) /* assume nit interface is active */
        niu_notify_8026(ifp, m0, mac_hdr_len, promisc);
    if (promisc)
        m_freem(m0);
    else {
        m_adj(m0, mac_hdr_len);
        if (ah->atm_pid1 == ATM_PID_LLC)
            m_freem(m0);
        else
            llc_data_ind(ifp, m0, src, mac_hdr_len, lp);
    }
    return 0;
}
/*

```

2/

if_atm.c
-43-

* svc_mac_input() handles aal payloads with SVC PDUs. Note, SVC PDUs
 * must fit in one mbuf. Hence the limitation for 4 ATM LANs for
 * 4.2BSD UNIX implementations. This could be changed by using
 * clusters for PDUs.

```

*/
svc_mac_input(vp, m)
struct vcte *vp;
struct mbuf *m;
{
    int len = m_len(m);

    if (len > MLEN) {
        svc_glob->svcstat.pdu_too_big++;
        m_freem(m);
        return 0;
    }
    m = m_pullup(m, len);
    if (m == 0) {
        svc_glob->svcstat.pdu_lost_nomem++;
        return 0;
    }
    ASSERT((mtod(m, int) & 3) == 0);
    if (m->m_next) {
        m_freem(m->m_next);
        m->m_next = 0;
    }
    svc_pdu(vp->vcte_pcif, mtod(m, caddr_t), len);
}

```

```

#include "../sys/domain.h"
extern struct domain svcdomain, pvcdomain;
#define ADDDOMAIN(x) { \
    extern struct domain x/**/domain; \
    x/**/domain.dom_next = domains; \
    domains = &x/**/domain; \
}

```

/*
 * svc_init() allocates global memory area for ATM signaling.
 */

```

extern int tr_buf_size;
struct vcte svc_vctes[VCTAB_SIZE];
int svc_init_count = 0;
struct ulptab svc_ulptab[NULPS];
struct atm_globs atm_globs;
struct tr_globs tr_globs;
struct svc_globs svc_globs;
char svc_e164_str[32];

```

```

svc_init()

```

82
if_atm.c
44

```

{
    struct vcte *vp;
    int      svc_mac_lmi(), svc_mac_input();

    if (svc_init_count)
        return;
    svc_glob->static_buf = svc_e164_str;
    svc_init_count = 1;
    bzero(&svc_glob->svcstat, sizeof(struct svcstat));
    svc_glob->vcte_free = svc_vctes;
    svc_glob->vcte_base = svc_vctes;
    svc_glob->ulptab = svc_ulptab;
    svc_glob->ulp_inuse = 0;
    for (vp = svc_vctes; vp < &svc_vctes[VCTAB_SIZE]; vp++)
        vp->vcte_next_cref = (struct vcte *) &vp[1];
    (-vp)->vcte_next_cref = 0;
    svc_glob->sig_ulp = ulp_register(LMI_LMI_ORG,
                                    LMI_LMI_PID,
                                    svc_mac_input, svc_mac_lmi, 0);
    svc_glob->svc_pcif =
        (struct pcif *) atm_alloc_bytes(sizeof(struct pcif) * NNIU);
    bzero((caddr_t) svc_glob->svc_pcif, sizeof(struct pcif) * NNIU);
    svc_glob->svc_pcifn = &svc_glob->svc_pcif[NNIU];
    svc_glob->svc_parms = svc_parms;

    atm_glob->atmif = (struct atmif *)
        atm_alloc_bytes(sizeof(struct atmif) * NATMS * NNIU);
    bzero((caddr_t) atm_glob->atmif,
        sizeof(struct atmif) * NATMS * NNIU);

    tr_init(tr_glob, tr_buf_size);
    ADDDOMAIN(pvc);
    ADDDOMAIN(svc);
    ADDDOMAIN(lfc);
    pvc_init();
}
#ifdef 0
atm_bzero(p, l)
    char *p;
{
    bzero(p, l);
}

atm_bcopy(s, d, l)
    char *s, *d;
{
    bcopy(s, d, l);
}

atm_bcmp(s, d, l)

```

83
if_atm.c
-45-

```

    char    *s, *d;
    {
        return bcmp(s, d, l);
    }
#endif

/*
 * svc_sched_timeout() is called to schedule svc_timeout() to be
 * called with the argument "pc" when the next signaling tick tock's.
 * On UNIX this is mapped onto timeout() converting svc_ms_per_tick
 * to Hertz. Presumably MS-DOS uses Avis based timeouts.
 */

svc_sched_timeout(pc)
    struct pcif *pc;
{
    int          svc_timeout();

    timeout(svc_timeout, (caddr_t) pc,
            (svc_ms_per_tick * hz) / 1000);
}

/*
 * svc_report_version_conflict() is called everytime a signaling PDU
 * with an unsupported version is received. It should report this
 * using the appropriate OS routines. For UNIX we printf to the
 * console no more than once every 15 seconds.
 */
struct timeval svc_last_conflict;

svc_report_version_conflict()
{
    extern struct timeval time;
    if ((svc_last_conflict.tv_sec + 15) < time.tv_sec) {
        printf("nl00: signaling protocol version conflict\n");
        svc_last_conflict.tv_sec = time.tv_sec;
    }
}

/*
 * atm_alloc_msg() allocates a msg buffer large enough for signal
 * PDUs.
 */
caddr_t
atm_alloc_msg()
{
    struct mbuf *m;

    m = m_get(M_DONTWAIT, MT_DATA);

```

84
if_atm.c
-46-

```

    if (!m)
        return 0;
    m->m_off = MMIOFF;
    m->m_len = 0;
    return mtod(m, caddr_t);
}

struct mbuf *atm_lastfm;

/*
 * atm_free_msg() frees message memory pointed to by cp.
 */
atm_free_msg(cp)
    char *cp;
{
    ASSERT(IVALID_VP((struct vcte *) cp));
    atm_lastfm = dtom(cp);
    m_freem(atm_lastfm);
}

/*
 * atm_alloc_bytes() is used to allocate memory for data structures
 * which are never freed, e.g., svc_pcif tables.
 */
caddr_t
atm_alloc_bytes(n)
{
    return kmem_alloc(n);
}

int
atm_trace_to_console = 0;

/*
 * Convert address to a hex byte string. The length of the address
 * is specified with the argument len.
 */
char *
atm_mac_sprintf(ap, len)
    u_char *ap;
{
    int i;
    char *cp = atm_glob->static_buf;
    static char digits[] = "0123456789abcdef";

    for (i = 0; i < len; i++) {
        *cp++ = digits[*ap >> 4];
        *cp++ = digits[*ap++ & 0xf];
        *cp++ = ':';
    }
    *--cp = 0;
    return atm_glob->static_buf;
}

```

25
if_atm.c
-47-

}

THIS PAGE BLANK (USPTO)

86
if atm_.h
-48-

```

/* if_atm.h
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */
#ifndef NIU_ATM_H
#define NIU_ATM_H Included

#include "bytes.h"
#include "unipdu.h"

/*
 * atm mac service interface (asi). This is the same as an ethernet
 * header so that upper layers can simply assume ATM is an ethernet.
 */
struct atmmsl {
    u_char    asi_dst[6];
    u_char    asi_src[6];
    u_short   asi_type;
};

/*
 * Structure of an ATM mac header for aal type 4, this is an 802.6
 * header.
 */
struct atm_header {
    struct atm_addr atm_dst;
    struct atm_addr atm_src;
    union {
        struct {
            u_int    mcb_pid:6;
            u_int    mcb_pad:2;
            u_int    mcb_delay:3;
            u_int    mcb_loss:1;
            u_int    mcb_crc:1;
            u_int    mcb_elen:3;
            u_int    mcb_pad1:16;
        }
        mcbbits;
        u_int    atm_mcb_long;
    }
    un_mcb;
};

#define atm_mcbbits un_mcb.atm_mcb_long
#define atm_elen    un_mcb.mcbbits.mcb_elen
#define atm_crc     un_mcb.mcbbits.mcb_crc
#define atm_loss    un_mcb.mcbbits.mcb_loss
#define atm_delay   un_mcb.mcbbits.mcb_delay
#define atm_pid     un_mcb.mcbbits.mcb_pid

```

87
if atm_.h
-49-

```
#define ATM_PID_LLC 1 /* protocol ID for LLC */
#define ATM_MCBITS_NOCRC 0x04000000 /* protocol id 1 */
#define ATM_HDR_LEN sizeof(struct atm_header)
#define ATM_PAD_SHIFT 24

/*
 * The only header extension defined is a return port address. The
 * length must be set to ATME_RPA_SIZE. Pad exists to get the 64 bit
 * address 64 bit aligned relative to the atm header.
 */

struct atm_header_ext {
    u_char    atme_len;
    u_char    atme_type;
    u_char    atme_pad[2]; /* need not be zeros (nnbz) */
    struct atm_addr atme_rpa; /* return port address */
};

#define ATME_RPA_TYPE 112 /* out of SMDS range */
#define ATME_RPA_BYTES sizeof(struct atm_header_ext)
#define ATME_RPA_WORDS ((sizeof(struct atm_header_ext)+3)/4)

/*
 * Callers to atm_data_req() must ensure atleast ATM_DATA_REQ_ROOM
 * bytes are available in front of the packet data.
 */
#define ATM_DATA_REQ_ROOM (ATM_HDR_LEN+LLC_SNAP_LEN+ATME_RPA_BYTES)

/*
 * multicast address structures are linked to atm_arptabs which are
 * marked ATF_MULTI. Such entries are not timed out, nor are they
 * freed when underlying VCs are released. atm_delete_lan() free's
 * the ATF_MULTI atm_arptab entries and atm_add_lan() &
 * atm_nlu_to_nlu() re-allocate them and re-initiate MC VCs for the
 * registered addresses.
 */
struct mcaddr {
    u_char    mc_enaddr[6]; /* multicast address */
    u_short   mc_count; /* reference count */
    struct aate *mc_at; /* multicast VC */
};

#define MCADDRMAX 64 /* multicast addr table length */
#define MCCOUNTMAX (32*1024-1) /* multicast addr max
 * reference count */

/*
 * atmif, one per atm lan, used by atm lan layer
 */
struct atmif {
    struct niu_arpcom *ati_ac; /* contains arp and ifnet
```

??

if atm_.h
-50-

```

    * structures */
    struct atmif *ati_next; /* linked off pcif structure */
    u_short      ati_state; /* basically do we know who
    * we are */
    u_short      ati_mld; /* mld used for multicast frames */
    u_short      ati_mcasts; /* max # multicasts circuits
    * configured */

    struct atm_addr ati_port;
    struct atm_addr ati_mac;
#define ac_mac      ati_mac.aa_byte[2]

    struct pcif *ati_pcif;
    struct aate *ati_arptab; /* set at initialization */
    int         ati_num_mcasts;
    struct mcaddr ati_mcaddrs[MCADDRMAX];
};

/* ati_state */
#define ATS_INACTIVE 0
#define ATS_ACTIVE 3

/*
 * global data structure for r/w variables and variables explicitly
 * initialized.
 */

#include "llc.h"

struct atm_globs {
    struct if_tr_hdr *lrb;
    struct atmif *atmif;
    int         atmifn;
    int         atmif_used;
    struct llc_snap llc_def;
    struct atm_addr atm_broadcast;
    struct atm_addr atm_null;
    struct ulptab *atm_ulp;
    int         atm_initialized;
    char        static_buf[32];
};

#ifdef RT68K
extern struct atm_globs atm_globs;
#define atm_glob (&atm_globs)
#else
#define atm_glob atm_get_glob()
struct atm_globs *atm_get_glob();
#endif

```

89

if atm_.h
-51-

#define LEN_FOR_MBUF_PTRS 0xf5560002

#define HASH_MULTICAST_ADDRESS(x) ((x)&0xff)

caddr_t atm_alloc_msg(), atm_alloc_bytes();

#define NATMS 4 /* max # of ATM lans per physical
* interface */

#define e160_ntoa svc_e164_ntoa/* these are really E.164 addresses */

#endif /* NIU_ATM_H */

90
if_niu.c
-52-

```
/* if_niu.c
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */

 * Description: This file contains the network interface portion of
 * the niu device driver.
 */
static char    sccsid[] = "%A%";

#ifdef KERNEL
#define KERNEL
#endif

#ifdef INET    /* only support internet addressing */
#define INET
#endif

#include "../sys/param.h"
#include "../sys/system.h"
#include "../sys/mbuf.h"
#include "../sys/socket.h"
#include "../sys/errno.h"
#include "../sys/ioctl.h"
#include "../sys/time.h"
#include "../sys/kernel.h"
#include "../sun4c/psl.h"

#include "../net/if.h"
#include "../net/netisr.h"
#include "../net/route.h"
#include "../net/if_arp.h"

#include "../sun/openprom.h"
#include "../sundev/mbvar.h"
#include "../sun4c/mmu.h"

#ifdef INET
#include "../netinet/in.h"
#include "../netinet/in_system.h"
#include "../netinet/in_var.h"
#include "../netinet/ip.h"
#endif

#include "debug.h"
#include "niu.h"
#include "unipdu.h"
```

91

if_niu.c
-53-

```

#include "atm.h"
#include "llc.h"
#include "svc.h"
#include "if_niuarp.h"
#include "atmarp.h"
#include "if_nlu.h"
#include "if_niulo.h"
#include "trace.h"
#include "../net/nit_if.h"
#include "snit.h"
int      aal_trace_enable = 1;

/* debugging and tracing stuff */
#define DL1      1
#define DL2      niu_debug>1
#define DL3      niu_debug>2
#define DL4      niu_debug>3
#define DL5      niu_debug>4
#define TL1      1
#define TL2      niu_trace>1
#define TL3      niu_trace>2
#define TL4      niu_trace>3
#define TL5      niu_trace>4

#define DRAIN TIME      4
/* the SIOCNIUDEBUG will set ALL debugging and tracing levels */
int      niu_debug = 1; /* conveys important driver
    * Information */
int      niu_trace = 1; /* show excessive detail of the
    * program stream */
extern int      arp_debug;
extern int      arp_trace;
extern int      drv_debug;
extern int      drv_trace;
extern int      dump_flag;

int      niu_mtu = 2000;
int      niu_unit_count = 0; /* total number of sw and hw
    * units installed */
struct niu_dev niu_info[NNIU]; /* network interface device structure */
int      niuoutput(), niusolctd();

/*
* Name:      niuoutput
*
* Input:      *ifp      - pointer to network interface to
* use. m      - mbuf pointer containing packet to be sent.
* dst      - ip address of destination.
*
* Output:      None.

```

92

if_niu.c
-54-

```

*
* Return: 0 - no error. Error -
* Unix error code.
*
* Description: This routine is called with a frame and a destination
* address. Appropriate address resolution is performed for the
* destination addresses until a VCI is obtained. The families
* supported are: AF_INET: An IP address is resolved using ARP
* into a 48 bit address. AF_UNSPEC: A 48 bit address is resolved
* into a 60 bit address and LLC/SNAP header is added. AF_NS: A 48
* bit address is resolved into a 60 bit address but no LLC/SNAP
* header is added. AF_CCITT: A 60 bit telephone number is resolved
* into a VCI using the port address to vci tables. AF_DL: A VCI
* is supplied. No resolution is performed. The VCI is in
* sockaddr_aal structure. AF_INET, AF_UNSPEC and AF_NS frames are
* encapsulated as per 802.6. Other address families are assumed to
* be encapsulated already.
*
* Note: The above families overload existing AF_XXX values. Also raw
* aal frames get queued on the first atm1an's ifnet send queue for
* lack of a better place to put them.
*/
int      niu_reset_on_full = 0, niu_auto_resets = 0;
int      willie_panic = 0;

niuoutput(ifp, m, dstin)
    struct ifnet *ifp;
    register struct mbuf *m;
    struct sockaddr *dstin;
{
    int      usetrailers, s, len, ifunit, rate;
    struct vcte *vp;
    struct ifqueue *ifq;
    struct sockaddr ldst;
    struct mbuf *mh;
    struct in_addr ldst;
    u_char    endest[6];
    struct niu_arpcorn *ac;
    struct aal_parms *ap;
    struct llc_snap *lc;
    int      iounit, error = 0;

    iounit = NIU_IFUNIT_TO_IOUNIT(ifp->if_unit);
    /* check if network is up */

    TR4(TL3, "niuoutput(%x, %x, %x, af=%d)\n",
        ifp, m, dstin, dstin->sa_family);
    if ((ifp->if_flags & IFF_UP) == 0 &&
        dstin->sa_family != AF_DL) {

```

93

if_niu.c
-55-

```

    m_freem(m);
    error = ENETDOWN;
    goto rtn;
}
ifunit = ifp->if_unit;
ac = &niu_arpcoms[ifunit];
s = splr(ip1tospl(niu_info[iouunit].priority));
#ifdef INET
if (dstin->sa_family == AF_INET) {
    ldst = ((struct sockaddr_in *) dstin)->sin_addr;
    DB2(DL2, "aa%d: dest = %s\n",
        ifunit, inet_ntoa(ldst));
    if (niu_arpresolve(ac, m, &ldst,
        ldst.sa_data, &usetrainers)) {
        goto rts;
    }
    ((struct ether_header *) ldst.sa_data)->ether_type =
        ETHERTYPE_IP;
    ldst.sa_family = AF_UNSPEC;
    dstin = &ldst;
} else
#endif
if (dstin->sa_family != AF_DLI &&
    dstin->sa_family != AF_NS &&
    dstin->sa_family != AF_UNSPEC) {
    printf("aa%d: can't handle af%d\n", ifp->if_unit,
        dstin->sa_family);
    m_freem(m);
    niu_info[iouunit].stats.errors++;
    error = EAFNOSUPPORT;
    goto rts;
}
/*
 * dstin.sa_data contains an ethernet header w/o a source adr
 * & type. (unless AF_DLI in which case we have a vcl...)
 */
if ((m->m_off & 0x3) || M_HASCL(m) || /* make room */
    (m->m_off - MMINOFF) < (ATM_DATA_REQ_ROOM +
        sizeof(struct aal_parms))) {
    if ((mh = m_get(M_DONTWAIT, MT_DATA)) == NULL) {
        m_freem(m);
        error = ENOBUFS;
        goto rts;
    }
    mh->m_len = 0;
    mh->m_off = MMAXOFF;
    mh->m_next = m;
    m = mh;
}
if (dstin->sa_family == AF_UNSPEC) {

```

94
if_niu.c
-56-

```

/* add LLC and SNAP */
lc = &(mtod(m, struct llc_snap *)[-1]);
*lc = atm_glob->llc_def;
lc->llc_type = ((struct atmmsl *) dstin->sa_data)->asl_type;
m->m_off -= sizeof(*lc);
m->m_len += sizeof(*lc);
}
len = 0;
for (mh = m; mh; mh = mh->m_next)
    len += mh->m_len;
if (dstin->sa_family == AF_UNSPEC ||
    dstin->sa_family == AF_NS) { /* get vci for mac
    *address */
    struct atm_header *ah = mtod(m, struct atm_header *);
    struct aate *aat;

    if ((ifp->if_flags & IFF_RUNNING) == 0) {
        m_freem(m);
        error = ENETDOWN;
        goto rts;
    }
    len += sizeof(*ah);
    ah->atm_dst_aa_long[0] = AAT_MAC << 28;
    bcopy(dstin->sa_data, &ah->atm_dst_aa_byte[2], 6);
    ah->atm_src = atm_glob->atmif[ifunit].atm_mac;
    ah->atm_mcbits = ((ATM_PID LLC << 2) +
        ((4 - (len & 3)) & 3)) << ATM_PAD_SHIFT;
    ah->atm_elen = 0;
    m->m_off -= sizeof(*ah);
    m->m_len += sizeof(*ah);
    aat = atm_find_at(&atm_glob->atmif[ifunit],
        dstin->sa_data);
    if (!aat || !!(vp = aat->aate_vcte)) {
        m_freem(m);
        error = EXDEV;
        goto rts;
    }
    if (vp->vcte_state < VCS_ESTAB) {
        atm_append_packet(vp, m);
        goto rts;
    }
} else {
    ASSERT(dstin->sa_family == AF_DLI);
    vp = ((struct sockaddr_aal *) dstin->saal_vcte);
    if (vp)
        ASSERT(VALID_VP(vp));
}
m->m_off -= sizeof(struct aal_parms);
m->m_len += sizeof(struct aal_parms);

```

95
if_niu.c
-57-

```

ap = mtod(m, struct aal_parms *);
ap->ap_mid = atm_glob->atmif[iunit].ati_mid;
ap->ap_vpcl = vp->vcte_ovpcl;
ap->ap_rate = vp->vcte_opeak_rate >> 4; /* from 1K bps to 16K
    * bps units */
if (vp->vcte_aal == 0)
    ap->ap_flags = AALP_RAW_CELL | AALP_CRC_NONE;
else
    ap->ap_flags = AALP_CRC_NONE;
/* iop, kludge till rev 2 fred 1/f with 960 gets implemented */
if ((!(vp->vcte_pclif->pc_flags & PCIF_NIU_TO_NIU)) &&
    vp->vcte_pclif->pc_sig->vcte_state == VCS_ACTIVE)
    ap->ap_flags |= AALP_ENABLE_XON_XOFF;
vp->vcte_opackets++;
ifp->if_opackets++;
/*
 * If multicast then frame must be single threaded so used
 * the atm lan index + 1 to indicate in which outbound queue
 * the frame should be placed.
 */
ap->ap_orderq = (vp->vcte_flags & VCTEF_MCAST_CLIENT) ?
    (int) (vp->vcte_atmif - atm_glob->atmif) + 1 :
    AALP_UNORDERED;
ap->ap_orderq = 4; /* iop, niu bug requires no more than
    * 1 vci per rate queue */
ap->ap_len = len + sizeof *ap;
/*
 * Place packet on interface transmit queue
 */
ifq = &niu_info[iunit].sendq;
if (IF_QFULL(ifq)) {
    DB0(DL2, "niuouput: interface q full\n");
    if (niu_reset_on_full || niu_info[iunit].type == NIU_TYPE_SW ||
        niu_info[iunit].board_id == NIU_REV3) {
        ASSERT(willie_panic == 0);
        while (m) {
            m_freem(m);
            IF_DEQUEUE(ifq, m);
            IF_DROP(ifq);
        }
        niu_auto_resets++;
        (*niu_info[iunit].reset) (iunit, 0);
        TR1(TL1, "aa%d auto reset\n", ifp->if_unit);
        printf("aa%d auto reset\n", ifp->if_unit);
        if (niu_info[iunit].type == NIU_TYPE_HW) {
            setup_rxbuf(&niu_info[iunit]);
        }
    } else {
        IF_DROP(ifq);
        m_freem(m);
    }
}

```

96
if_niu.c
-58-

```

    }
    error = ENOBUFS;
} else {
    TR1(TL3, "%d on queue: ", ifp->if_snd.ifq_len);
    IF_ENQUEUE(ifq, m);
}
(*niu_info[iounit].sendpkt) (iounit);
rts:
    splx(s);
rtn:
    TR3(TL3, "niuoutput->%d, %d q %d d\n",
        error, ifq->ifq_len, ifq->ifq_drops);
    return error;
}

/*
 * send mbuf chain m on physical interface pc over VC vp using the
 * aal associated with that vp. mid is the multiplex id for aal 3/4.
 * It is ignored for aal 5.
 */
vcoutput(vp, m, mid)
    struct mbuf *m;
    struct vcte *vp;
{
    int s, len;
    struct ifqueue *ifq;
    struct mbuf *mh;
    struct aal_parms *ap;
    int iounit = vp->vcte_pcif->pc_num;
    int error = 0;

    ASSERT(VALID VP(vp));
    s = splr(ipltostr(niu_info[iounit].priority));

    if ((m->m_off & 0x3) || M_HASCL(m) ||
        (m->m_off - MMINOFF < sizeof(struct aal_parms)) {
        if ((mh = m_get(M_DONTWAIT, MT_DATA)) == NULL) {
            m_freem(m);
            error = ENOBUFS;
            goto rts;
        }
        mh->m_len = 0;
        mh->m_off = MMAXOFF;
        mh->m_next = m;
        m = mh;
    }
    m->m_off -= sizeof(struct aal_parms);
    m->m_len += sizeof(struct aal_parms);
    ap = mtod(m, struct aal_parms *);
    ap->ap_mid = mid;

```

97

if_niu.c
-59-

```

ap->ap_vpci = vp->vcte_ovpci;
ap->ap_rate = vp->vcte_opeak_rate >> 4; /* from 1K bps to 16K
    * bps units */
if (vp->vcte_aal == 0)
    ap->ap_flags = AALP_RAW_CELL | AALP_CRC_NONE;
else
    ap->ap_flags = AALP_CRC_NONE;
if ((!(vp->vcte_pcif->pc_flags & PCIF_NIU_TO_NIU)) &&
    vp->vcte_pcif->pc_sig &&
    vp->vcte_pcif->pc_sig->vcte_state == VCS_ACTIVE)
    ap->ap_flags |= AALP_ENABLE_XON_XOFF;
vp->vcte_opackets++;
/*
 * if multicast then frame must be single threaded so used
 * the atm lan index + 1 to indicate in which outbound queue
 * the frame should be placed.
 */
#ifdef 0
    ap->ap_orderq = (vp->vcte_flags & VCTEF_MCAST_CLIENT) ?
        (int) (vp->vcte_atmif - atm_glob->atmif) + 1 :
        AALP_UNORDERED;
#endif
ap->ap_orderq = 4; /* lop, niu bug requires no more than
    * 1 vci per rate queue */
for (len = 0, mh = m; mh; mh = mh->m_next)
    len += mh->m_len;
ap->ap_len = len;
/*
 * Place packet on interface transmit queue
 */
ifq = &niu_info[iounit].sendq;
if (IF_QFULL(ifq)) {
    DEB(DL2, "vcoutput: Interface q full\n");
    if (niu_reset_on_full ||
        niu_info[iounit].type == NIU_TYPE_SW ||
        niu_info[iounit].board_id == NIU_REV3) {
        ASSERT(willie_panic == 0);
        while (m) {
            m_freem(m);
            IF_DEQUEUE(ifq, m);
            IF_DROP(ifq);
        }
        niu_auto_resets++;
        (*niu_info[iounit].reset) (iounit, 0);
        TR1(TL1, "niu%d auto reset\n", iounit);
        printf("niu%d auto reset\n", iounit);
        if (niu_info[iounit].type == NIU_TYPE_HW) {
            setup_rxbuf(&niu_info[iounit]);
        }
    }
} else {

```

if_niu.c
-60-

```

        IF_DROP(ifq);
        m_freem(m);
    }
    error = ENOBUFS;
} else {
    TR1(TL1, "%d on queue: ", ifq->ifq_len);
    IF_ENQUEUE(ifq, m);
}
(*niu_info[lount].sendpkt) (lount);
rts:
    splx(s);
    TR3(TL1, "vcount->%d, %d q %d d\n",
        error, ifq->ifq_len, ifq->ifq_drops);
    return error;
}

```

```

int      niu_esr = 0;
niu_restart_sends(unit)
{
    struct atmif *atp;

    if (!niu_esr)
        return;
    atp = svc_glob->svc_pcif[unit].pc_atmif;
    while (atp) {
        if (((struct ifnet *) atp->ati_ac)->if_snd.ifq_len)
            (*niu_info[unit].sendpkt)(unit);
        atp = atp->ati_next;
    }
}

```

```

/*
 * Name: niuioctl
 *
 * Input: *ifp - pointer to network interface to use. cmd
 * - command requested. *data - data associated with the command.
 *
 * Output: *data - data may be filled in by certain commands.
 *
 * Return: 0 - no error. Error - Unix error code.
 *
 * Description: This is the network interface ioctl routine. An ifreq
 * structure must be used to access this routine.
 */
niuioctl(ifp, cmd, data)
    register struct ifnet *ifp;
    int      cmd;
    caddr_t   data;

```

99

if_niu.c
-61-

```

{
    int          error = 0, l, s, svc_timeout();
    int          ifunit = ifp->if_unit;
    extern int    atmarp_timeo1, atmarp_timeo2;
    struct ifreq  *ifr = (struct ifreq *) data;
    struct ifaddr *ifa = (struct ifaddr *) data;
    struct niu_arpcom *ac = (struct niu_arpcom *) ifp;
    struct db_info *dbp;
    struct mcaddr *mca;
    struct atmif  *atp;

    TR2(TL3, "aa%d(%x): niusoloctl entered\n", ifunit, cmd);

    switch (cmd) {

    case SIOCSIFADDR:
        /* set the Interface Ip address */
        TR1(TL4, "aa%d: loctl SIOCSIFADDR\n", ifunit);
        switch (ifa->ifa_addr.sa_family) {
        #ifdef INET
            case AF_INET:
                niu_arpcoms[ifunit].ac_ipaddr =
                    IA_SIN(ifa->sin_addr);
                ifp->if_flags |= IFF_UP;
                break;
        #endif
            default:
                break;
        }
        break;

    case SIOCSIFFLAGS:
        /* set Interface flags */
        TR3(TL4, "aa%d: loctl SIOCSIFFLAG (%d)flag=%d\n",
            ifunit, ifp->if_flags, ifr->ifr_flags);
        ifp->if_flags = ifr->ifr_flags;
        if (ifp->if_flags & IFF_UP) {
            struct niu_dev *niu;
            niu = &niu_info[NIU_IFUNIT TO IOUNIT(ifp->if_unit)];
            if (niu->type == NIU_TYPE_HW) {
                setup_rxbuf(niu);
            }
        }
        break;

    case SIOCGIFFLAGS:
        /* get Interface flags */
        TR2(TL4, "aa%d: loctl SIOCGIFFLAG flag=%d\n",
            ifunit, ifp->if_flags);
        ifr->ifr_flags = ifp->if_flags;

```

100
if_niu.c
-62-

```

break;

case SIOCGETPORT:
    /* get port address */
    TR3(TL4, "aa%d: ioctl SIOCGETPORT port=%8x%8x\n", ifunit,
        atm_glob->atmif[ifunit].atl_port.aa_long[0],
        atm_glob->atmif[ifunit].atl_port.aa_long[1]);
    ifr->ifr_addr.sa_family = AF_CCITT;
    bcopy(&atm_glob->atmif[ifunit].atl_port,
        ifr->ifr_addr.sa_data,
        sizeof(struct atm_addr));
    break;

case SIOCNIUDEBUG:
    /* set debug level for the entire niu device */
    dbp = (struct db_info *) ifr->ifr_data;
    niu_debug = dbp->niu_debug; /* network interface */
    niu_trace = dbp->niu_trace;
    arp_debug = dbp->arp_debug; /* arp */
    arp_trace = dbp->arp_trace;
    drv_debug = dbp->drv_debug; /* /dev/niu */
    drv_trace = dbp->drv_trace;
    DB3(DL2, "aa%d: ioctl SIOCSNIUDEBUG niu=%d %d\n",
        ifunit, niu_debug, niu_trace);
    DB3(DL2, "aa%d: ioctl SIOCSNIUDEBUG arp=%d %d\n",
        ifunit, arp_debug, arp_trace);
    DB3(DL2, "aa%d: ioctl SIOCSNIUDEBUG drv=%d %d\n",
        ifunit, drv_debug, drv_trace);
    break;

case SIOCGIFADDR:
case SIOCGMACADDR:
    /* get the interface ip address */
    TR1(TL2, "aa%d: ioctl SIOCGIFADDR\n", ifunit);
    bcopy(&niu_arpcoms[ifunit].ac_atmif->atl_mac.aa_byte[2],
        ifr->ifr_addr.sa_data, 6);
    break;

case SIOCSMACADDR:
    /* set the interface ip address */
    TR1(TL2, "aa%d: ioctl SIOCSIFADDR\n", ifunit);
    bcopy(ifr->ifr_addr.sa_data,
        &niu_arpcoms[ifunit].ac_atmif->atl_mac.aa_byte[2], 6);
    break;

case SIOCGAETIME01:
    bcopy(&atmarp_timeo1, &ifr->ifr_metric, sizeof(int));
    break;

case SIOCSAETIME01:
    bcopy(&ifr->ifr_metric, &atmarp_timeo1, sizeof(int));
    break;

case SIOCGAETIME02:

```

/o/
if_niu.c
-63-

```

    bcopy(&atmarp_timeo2, &ifr->lfr_metric, sizeof(int));
    break;
case SIOCSAATIMEO2:
    bcopy(&ifr->lfr_metric, &atmarp_timeo2, sizeof(int));
    break;
case SIOCTIMEOUT:
    if ((atp = atm_glob->atmif) == 0) {
        error = ENXIO;
        break;
    }
    s = splnet();
    untimeout(svc_timeout, atm_glob->atmif[ifunit].ati_pcif);
    svc_timeout(atm_glob->atmif[ifunit].ati_pcif);
    for (; atp < &atm_glob->atmif[atm_glob->atmif_used];
        atp++) {
        if (atp->ati_state == ATS_INACTIVE)
            continue;
        atm_arptimer(atp);
    }
    splx(s);
    break;
case SIOCADDMULTI:
    TR1(TL4, "aa%d: ioctl SIOCADDMULTI\n", ifunit);
    error = niu_addmulti(ifp, ifr->ifr_addr.sa_data);
    break;
case SIOCDELMULTI:
    TR1(TL4, "aa%d: ioctl SIOCADDMULTI\n", ifunit);
    error = niu_delmulti(ifp, ifr->ifr_addr.sa_data);
    break;
case SIOCSPROMISC:
    ifp->if_flags ^= IFF_PROMISC;
    printf("aa%d promiscuous %sabled\n", ifunit,
        ifp->if_flags & IFF_PROMISC ? "en" : "dis");
    break;
case SIOCGSTATE:
    /* get signaling vc state */
    TR2(TL4, "aa%d: ioctl SIOCGSTATE state=%d\n", ifunit,
        atm_glob->atmif[ifunit].ati_pcif->pc_sig->vcte_state);
    ifr->lfr_metric =
        atm_glob->atmif[ifunit].ati_pcif->pc_sig->vcte_state;
    break;
case SIOCSSTATE:
    /* get signaling vc state */
    TR2(TL4, "aa%d: ioctl SIOCGSTATE state=%d\n", ifunit,
        atm_glob->atmif[ifunit].ati_pcif->pc_sig->vcte_state);
    if (ifr->lfr_metric < VCS_INACTIVE ||
        ifr->lfr_metric > VCS_ACTIVE)
        error = EINVAL;
    else
        svc_new_state(atm_glob->atmif[ifunit].ati_pcif->pc_sig,

```

102
if_niu.c
-64-

```

        ifr->ifr_metric);
    break;
default:
    DB2(DL2, "aa%d: ioctl bad command=%x\n",
        ifunit, cmd);
    error = EINVAL;
}
return (error);
}

/*
 * Find a multicast entry in the multicast filter for atm lan, atp.
 */

struct mcaddr *
niu_findmulti(atp, mac)
    struct atmif *atp;
    u_char *mac;
{
    int i;

    for (i = 0; i < atp->ati_num_mcasts; i++)
        if (bcmp(atp->ati_mcaddrs[i].mc_enaddr, mac, 6) == 0)
            return &atp->ati_mcaddrs[i];
    return 0;
}

/*
 * Add a multicast address to multicast filter for atm lan, atp.
 */
niu_addmulti(ifp, mac)
    struct ifnet *ifp;
    u_char *mac;
{
    int i, s, error;
    struct aate *at;
    struct mcaddr *mc;
    struct atmif *atp = ((struct niu_arpcom *) ifp)->ac_atmif;

    if ((mac[0] & 0x1) == 0)
        return EINVAL; /* not a multicast address */
    mc = niu_findmulti(atp, mac);
    s = splimp0();
    if (mc) {
        if (mc->mc_count < MCCOUNTMAX) {
            mc->mc_count++;
            splx(s);
            return 0;
        } else {
            splx(s);
            return ENOSPC;
        }
    }
}

```

/03

if_niu.c
-65-

```

    }
}
if (atp->ati_num_mcasts == MCADDRMAX) {
    splx(s);
    return ENOSPC;
}
mc = &atp->ati_mcaddrs[atp->ati_num_mcasts];
mc->mc_count = 1;
bcopy(mac, mc->mc_enaddr, 6);
at = mc->mc_at = atm_find_at(atp, mac);
if (at == 0) {
    splx(s);
    return ENOSPC;
}
at->aate_flags |= ATF_MULTI;
atp->ati_num_mcasts++;
splx(s);
return 0;
}

/*
 * Delete a multicast address from multicast filter for atm lan, atp.
 */
niu_delmulti(ifp, mac)
    struct ifnet *ifp;
    u_char *mac;
{
    struct mcaddr *mc;
    int i, s;
    struct aate *at;
    struct atmif *atp = ((struct niu_arpcorn *) ifp)->ac_atmif;

    mc = niu_findmulti(atp, mac);
    if (mc == 0)
        return ENXIO;
    s = splimp();
    if (--mc->mc_count > 0) {
        splx(s);
        return 0;
    } else if (at = mc->mc_at) {
        ASSERT(at->aate_flags & ATF_MULTI);
        at->aate_flags &= ~ATF_MULTI;
        atm_aate_free(at);
    }
    bcopy(&atp->ati_mcaddrs[atp->ati_num_mcasts], mc, sizeof(*mc));
    splx(s);
    return 0;
}

```

104
if_niu.c
-66-

```

/*
 * go steal an ethernet's low order 2 bytes and append it two
 * Adaptive's IEEE prefix.
 */
unsigned int  def_enaddr[2] = {0x0080b2e0, 0x00010000};

niu_get_enaddr(unit, enaddr)
    u_int      unit;
    u_char     *enaddr;
{
    struct ifnet *ifp;
    extern struct ifnet *ifnet;

    bcopy(def_enaddr, enaddr, 6);
    for (ifp = ifnet; ifp; ifp = ifp->if_next)
        if (ifp->if_mtu == 1500) {
            enaddr[3] = (((struct niu_arpcom *) ifp)->ac_enaddr[3] & 0x1f) | 0xe0;
            enaddr[4] = ((struct niu_arpcom *) ifp)->ac_enaddr[4];
            enaddr[5] = ((struct niu_arpcom *) ifp)->ac_enaddr[5];
            break;
        }
    enaddr[0] = (u_char) unit << 1;
}

/*
 * This routine just looks up the input vci and dispatches the frame
 * to the appropriate input routine based upon vci. Signaling and
 * raw user access does not necessarily use 802.6 framing.
 */
int
deliver_packet(unit, m0, vci)
    int      unit;
    struct mbuf *m0;
    u_short  vci;
{
    struct vcte *vp;
    struct pcif *pc;
    int      s;
    int      plen;

    plen = m_len(m0);

    aal_trace_m(m0, plen, 1, vci);
    s = splimp0;
    pc = &svc_glob->svc_pcif[unit];
    if (pc->pc_raw_vp) {
        pc->pc_raw_vp->vcte_ipackets++;
        pvc_input(pc->pc_raw_vp, m0);
    } else if (vp = vpcl_to_vcte(pc, vci)) {
        if (VCS_DATA_IND_OK & VCS_TO_VMASK(vp->vcte_state)) {

```

105

if_niu.c
-67-

```

    ASSERT(VALID_ULP(vp->vcte_ulp));
    vp->vcte_ipackets++;
    (*vp->vcte_ulp->ulp_data)(vp, m0);
} else
    m_freem(m0);
} else if (svc_send_releases && pc->pc_sig) {
    struct release *pdu;
    m_freem(m0);
    pdu = (struct release *) atm_alloc_msg0();
    pdu->lmi_proto = LMI_PROTOCOL;
    pdu->lmi_pdu_type = TDU_INVALID_PDU;
    pdu->lmi_cref_type = LMI_CREFTYPE_PVC;
    pdu->lmi_cref_value = vci;
    LMI_SET_ELEMENT(&pdu->lmi_cause, LMI_RELEASE_CAUSE,
        VCI_UNACCEPTABLE);
    svc_xdu(pc, 0, pdu, sizeof *pdu);
}
splx(s);
}

/*
 * niu_snitify() makes a copy of m0, converts the 802.6/SNAP header
 * into an ethernet header and calls snit_intr().
 */
struct nit_if niu_nit;
u_shortenet_hdr[7];

niu_snitify_8026(ifp, m, hlen, promisc)
    struct ifnet *ifp;
    struct mbuf *m;
{
    int adj;
    u_char *sp; /* start of destination adr in 802.6
        * header */

    ASSERT(hlen >= 20);
    if (m->m_len < hlen + 3)
        return; /* not enough for llc */
    sp = mtod(m, u_char *);
    bcopy(&sp[2], enet_hdr, 6);
    bcopy(&sp[2 + 8], &enet_hdr[3], 6);
    if (sp[hlen] == (u_char) 0xaa) {
        bcopy(&sp[hlen + 6], &enet_hdr[6], 2);
        adj = hlen + 8;
    } else {
        bcopy(&sp[hlen], &enet_hdr[6], 2);
        adj = hlen + 3;
    }
    if (m->m_len < adj)

```

106

if_niu.c

-68-

```

    return;
    m->m_len -= adj;
    m->m_off += adj;
    niu_nit.nif_header = (caddr_t) enet_hdr;
    niu_nit.nif_hdrlen = 14;
    niu_nit.nif_bodylen = m_len(m) - 14;
    niu_nit.nif_promisc = promisc;
    snit_intr(ifp, m, &niu_nit);
    m->m_len += adj;
    m->m_off -= adj;
}

int      dump_len = 64;
int      dump_limit = 0;

dump_frame(s, dp, words)
    char      *s;
    int       *dp;
    int       words;
{
    if (dump_limit == 0)
        return;
    printf("%s ", s);
    if (words > dump_limit)
        words = dump_limit;
    while (words--)
        printf("%x ", *dp++);
    printf("\n");
}

int      dumpbuf[64];
int      dumplen = 64;

dump_chain(s, m)
    char      *s;
    struct mbuf *m;
{
    int      left = m_len(m);

    if (left > dumplen)
        left = dumplen;
    m_copydat(m, (char *) dumpbuf, left);
    left = (left + 3) / 4;
    dump_frame(s, dumpbuf, left);
}

m_len(m)
    struct mbuf *m;
{
    int      len;

```

/07

if_nlu.c
-69-

```

    for (len = 0; m; m = m->m_next)
        len += m->m_len;
    return len;
}

atm_arploctl(ifunit, cmd, data)
    int      cmd;
    caddr_t   data;
{
    struct arpreq *ar = (struct arpreq *) data;
    struct aate *at;
    int      s, error = 0;

    if (ar->arp_pa.sa_family != AF_UNSPEC ||
        ar->arp_ha.sa_family != AF_CCITT)
        return (EAFNOSUPPORT);

    s = splimp0;
    at = atm_arptab_look(&atm_glob->atmif[ifunit],
                        ar->arp_pa.sa_data);
    if (at == NULL)
        error = ENXIO;
    else if (ar->arp_pa.sa_data[0] & 0x01)
        error = EINVAL;
    else if (cmd == SIOCDDARP)
        atm_aate_free(at);
    splx(s);
    return error;
}

calc_mlen(m)
    struct mbuf *m;
{
    int      len = 0;

    TR0(TL2, "calc_mlen: called\n");

    while (m) {
        len += m->m_len;
        m = m->m_next;
    }
    DB1(DL3, "calc_mlen: len=0x%x\n", len);
    return (len);
}

m_copydat(m, buf, len)
    struct mbuf *m;
    char *buf;
    int      len;
{

```

108

if_niu.c
-70-

```
int j;
while (len && m) {
    j = len;
    if (j > m->m_len)
        j = m->m_len;
    if (!) {
        bcopy(mtod(m, caddr_t), buf, j);
        buf += j;
        len -= j;
    }
    m = m->m_next;
}
return len;
}
```

```
aal_trace_m(m, tlen, in, vci)
struct mbuf *m;
{
    int s;
    if (!aal_trace_enable)
        return;
    s = spl7();
    atm_trace_buf(m, m_copydat, IF_TRACE_LOG, tlen, in, vci);
    splx(s);
}
```

109

if_niu.h
-71-

```

/* if_niu.h
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */
/* static char sccsid[] = "@(#) if_niu.h 1.12@(#)"; */
/*
 * The aal Interface is implemented as messages send (a)synchronously
 * between the MAC and aal layers. The aal_parms structure preceeds
 * frames transmitted and received. It is the complete interface
 * between the aal layer and the aal user. Rather than define some
 * VC parameters at circuit setup time and pass other per frame
 * parameters with each frame, all parameters are passed with each
 * frame. Thus, at the loss of some performance, the interface
 * between the H/W NIU (and 960 OS) and the host is simplified.
 *
 * To do: Average rate metering parameters will be added when we figure
 * out how to use them. frame level CRC should be specified but
 * current chips have a mode bit for CRC.
 */
struct aal_parms {
    vpci_t      ap_vpci; /* vpci to be operated upon */
    u_short     ap_mid; /* mid to use with frame */
    u_short     ap_len; /* packet length (excluding aal_parms
        * and */
    /* pad bytes if AALP_CRC_SMDS) */
    u_short     ap_rate; /* burst rate for frame divided by
        * 1024 */
    u_char      ap_orderq; /* Identifies an ordered send
        * queue. Frames */
    /* with the same orderq may not be interleaved. */
    /* AALP_UNORDERED indicates no restrictions */
    u_char      ap_flags; /* AALP_CRC_xxx */
};

/* ap_crc32 values */
#define AALP_CRC_NONE      0
#define AALP_CRC_ADAPTIVE  1
#define AALP_CRC_SMDS      2
#define AALP_RAW_CELL      4
#define AALP_ENABLE_XON_XOFF 8 /* enable xon/xoff higher
    * layer */
#define AALP_LOOP_VCI      0x10 /* loop rcv frames at 960 */

/* ap_rate value to specify maximum link rate */
#define AALP_MAX_RATE      (~0) /* all one's */
/* ap_orderq value if frame has no ordering constraints */
#define AALP_UNORDERED      0

/*
 * niuoutput() sockaddr used for raw aal access with AF_DLJ.

```

//0

if_niu.h
-72-

* saal_vcte must reference a valid vcte.
*/

```
struct sockaddr_aal {
    u_short    saal_family;
    u_short    saal_pad2;
    struct vcte *saal_vcte;
    char       saal_pad6[6];
};
```

```
struct niu_desc {
    u_char      status;
    u_char      niual;
    u_short     reserved;
    u_int       pkt_addr;
    u_short     size;
    u_short     vcl;
    u_int       chain_ptr;
};
```

/* used for SIOCNIUDEBUG ioctl */

```
struct db_info {
    char        niu_debug;
    char        niu_trace;
    char        arp_debug;
    char        arp_trace;
    char        drv_debug;
    char        drv_trace;
};
```

/* MTU size */
#define AAMTU

9188

/* receive control register */
#define RCNTL_REG 0
#define RCNTL_IDLE_INTR 0x80 /* rx fill interrupt */
#define RCNTL_FILL_INTR 0x40 /* rx idle interrupt */
#define RCNTL_PASS_IDLE 0x04 /* go through idle on every
* cell */
#define RCNTL_STOP_IDLE 0x02 /* stop on idle */
#define RCNTL_RESET 0x01 /* reset rx fifo, abort cell,
* spill mode */
#define RCNTL_MASK 0xc7 /* bits 3-5 unused */

/* receive status register 1 */
#define RSTAT1_REG 1
#define RSTAT1_LIGHT 0x80 /* rx fiber light present */
#define RSTAT1_FIFO_HALF 0x40 /* rx fifo half flag */
#define RSTAT1_FIFO_FULL 0x20 /* rx fifo full flag */
#define RSTAT1_FIFO_EMPTY 0x20 /* rx fifo empty flag */

```

///
if_niu.h
-73-

```

```

#define RSTAT1_VIOLATION 0x04 /* rx violation */
#define RSTAT1_MASK 0xec /* bits 0,1,4 unused */

/* receive status register 2 */
#define RSTAT2_REG 2
#define RSTAT2_IDLE 0x80 /* rx is idle */
#define RSTAT2_FIFO_OVR 0x40 /* rx fifo overflow */
#define RSTAT2_CMD_OVR 0x20 /* rx command overflow */
#define RSTAT2_CMD_RECV 0x10 /* rx command received */
#define RSTAT2_COMMAND 0x0f /* rx command, 4 bits */
#define RSTAT2_INTR_MASK 0xd0 /* rx interrupt mask */

/* transmit control register */
#define TCNTL_REG 3
#define TCNTL_RESET 0x80 /* tx reset */
#define TCNTL_LOAD 0x40 /* load tx fifo */
#define TCNTL_SOC_ENBL 0x20 /* start of cell enable */
#define TCNTL_ENABLER 0x10 /* enable send from fifo */
#define TCNTL_COMMAND 0x0f /* tx command */

/* transmit status register */
#define TSTAT_REG 4
#define TSTAT_FIFO_FULL 0x80 /* tx fifo full */
#define TSTAT_FIFO_HALF 0x40 /* tx fifo half */
#define TSTAT_FIFO_EMPTY 0x20 /* tx fifo empty */
#define TSTAT_MASK 0xe0 /* bits 0-4 unused */

#define MAX_INTR_TIME 200

/* dma controller control/status */
#define DMAC_INT_PEND 0x00000001 /* interrupt pending */
#define DMAC_ERR_PEND 0x00000002 /* error pending */
#define DMAC_DRAINING 0x0000000c /* draining D cache */
#define DMAC_INT_EN 0x00000010 /* interrupt enable */
#define DMAC_FLUSH 0x00000020 /* flush buffer */
#define DMAC_SLAVE_ERR 0x00000040 /* slave error */
#define DMAC_RESET 0x00000080 /* reset DMA */
#define DMAC_WRITE 0x00000100 /* 1 = memory write; 0 =
    * memory read */
#define DMAC_EN_DMA 0x00000200 /* enable dma */
#define DMAC_EN_CNT 0x00002000 /* enable counter */
#define DMAC_TC 0x00004000 /* terminal count */
#define DMAC_ALE_AS 0x00100000 /* 1 = addr latch enb; 0
    * = addr strobe */
#define DMAC_LANCE_ERR 0x00200000 /* E channel error */
#define DMAC_FASTER 0x00400000 /* fast access for D
    * channel */
#define DMAC_TCI_DIS 0x00800000 /* TC interrupt disable */
#define DMAC_EN_NEXT 0x01000000 /* enable next */
#define DMAC_DMA_ON 0x02000000 /* DMA on */

```

//2

if_niu.h
-74-

```

#define DMAC_A_LOADED    0x04000000 /* address loaded */
#define DMAC_NA_LOADED   0x08000000 /* next address loaded */
#define DMAC_DEV_ID      0x00000000 /* device id */

#define DMAC_INTR_MASK    0x00000003 /* DMAC Interrupt
    * pending mask */

/* dma address */
#define DMAC_ADDR_REG      6

/* dma next address */
#define DMAC_ADDRNXT_REG   7

/* dma count */
#define DMAC_COUNT_REG     8

/* dma next count */
#define DMAC_CNTNXT_REG    9

#define SW_NUM_SWREGS     10 /* number of registers on sw
    * niu */
#define NUM_SWREGS        3 /* number of registers on sw
    * niu */
#define NUM_SWINTR        1 /* number of interrupts on sw
    * niu */

struct niu_addr_reg {
    u_char    *rcntl_reg; /* receive control register */
    u_char    *rstat1_reg; /* receive status 1 register */
    u_char    *rstat2_reg; /* receive status 2 register */
    u_char    *tcntl_reg; /* transmit control register */
    u_char    *tstat_reg; /* transmit status register */
};

struct niu_value_reg {
    u_char    rcntl_reg; /* receive control register */
    u_char    rstat1_reg; /* receive status 1 register */
    u_char    rstat2_reg; /* receive status 2 register */
    u_char    tcntl_reg; /* transmit control register */
    u_char    tstat_reg; /* transmit status register */
};

struct hw_niu_reg {
    u_long    *dma_reg; /* LSI dma status register */
    u_short   *attn_reg; /* niu attention register */
    u_long    *base_reg; /* niu base register */
    u_short   *intr_reg; /* niu interrupt acknowledge
    * register */
    u_short   *lock_reg; /* niu dma lockout register */
    u_long    dma_value; /* local copy of dma status

```

113
if niu.h
-75-

```

    * register */
    u_short   attn_value; /* local copy of niu
    * attention register */
    u_long    base_value; /* local copy of niu base
    * register */
};

struct dmac_addr_reg {
    u_long    *status_reg; /* status control register */
    u_long    *addr_reg; /* address register */
    u_long    *next_address_reg; /* next address register */
    u_long    *count_reg; /* count register */
    u_long    *next_count_reg; /* next count register */
};

#define NUM_DESC 1 /* up to 1 descriptors in
    * chain */
typedef struct {
    caddr_t    dma_addr;
    int        size;
} DMA_DESC_BUF;

struct dmac_value_reg {
    u_long    status_reg; /* status control register */
    u_long    addr_reg; /* address register */
    u_long    next_address_reg; /* next address register */
    u_long    count_reg; /* count register */
    u_long    next_count_reg; /* next count register */
};

struct niu_stats {
    u_long    ip_opkts; /* number tx ip packets */
    u_long    ip_ipkts; /* number rx ip packets */
    u_long    arp_opkts; /* number tx ip packets */
    u_long    arp_ipkts; /* number rx ip packets */
    u_long    drv_opkts; /* number tx driver packets */
    u_long    drv_ipkts; /* number rx driver packets */
    u_long    crc_errors; /* total number crc errors */
    u_long    errors; /* total number misc errors */
    u_long    allocd_failed; /* number of alloc_desc
    * failures */
    u_long    finddesc_failed; /* number of mismatched
    * tags */
};

/* packet direction */
#define NIU_RECEIVE 0 /* host receiving packets
    * from niu */
#define NIU_TRANSMIT 1 /* host trasmitting packets
    * to niu */

```

114
if_niu.h
-76-

```
typedef struct {
    int      in_use;
    int      cmd_tag;
    struct mbuf *m;
    int      num_desc;
    caddr_t  desc_addr;
    DMA_DESC_BUF *desc_ptr;
    caddr_t  data_addr;
}          DMA_DESC;
#define NUM_DMA_DESC 11

#define COMMAND_SIZE 16

#define NO_COMMAND 0
#define RESET_CMD 1
#define STATUS_CMD 2
#define CLR_STATS_CMD 3
#define RX_DATA_CMD 4
#define TX_DATA_CMD 5
#define CLR_INTR_CMD 6
#define RESET_Q_CMD 7
#define WORK_AROUND_CMD 8
#define BOARD_ID_CMD 9

#define CMD_INTR_OFF 0x00
#define CMD_INTR_ON 0x01
#define CMD_CRC_MASK 0x06 /* frame level crc */
#define CMD_CRC_ADAPTIVE 0x02
#define CMD_CRC_SMDS 0x04
#define CMD_CRC_NONE 0x00
#define CMD_AAL_MASK 0x18
#define CMD_AAL4 0x00 /* default is aal4 */
#define CMD_AAL5 0x08 /* not yet implemented */
#define CMD_AAL_RAW 0x18 /* send raw cell ala s/w niu */
#define CMD_ENABLE_XON_XOFF 0x20 /* enable xon/xoff */
#define CMD_LOOP_VCI 0x40 /* loop rcv frames at 960 */

typedef struct {
    u_char      param[COMMAND_SIZE - 4];
    u_short     tag;
    u_char      flags;
    u_char      command;
}              COMMAND;

typedef struct {
    u_char      param[COMMAND_SIZE - 8];
    u_int       board_id;
    u_short     tag;
    u_char      flags;
}
```

//5
if_niu.h
-77-

```
    u_char    command;
}    BID_CMD;
```

```
typedef struct {
    caddr_t    dma_addr;
    u_short    vci;
    u_short    mid;
    u_short    size;
    u_char     order_q;
    u_char     rate_q;
    u_short    tag;
    u_char     flags;
    u_char     command;
}    RX_CMD;
```

```
typedef struct {
    caddr_t    dma_addr;
    u_short    vci;
    u_short    mid;
    u_short    size;
    u_char     order_q;
    u_char     rate_q;
    u_short    tag;
    u_char     flags;
    u_char     command;
}    TX_CMD;
```

```
#define CMD_Q_SIZE    NUM_DMA_DESC
#define START_CMD_Q(q)    (&((q)->cmd_q[0]))
#define CUR_CMD_Q(q)    (&((q)->cmd_q[(q)->cmd_elem]))
#define END_CMD_Q(q)    (&((q)->cmd_q[CMD_Q_SIZE-1]))
#define NEXT_CMD_Q(elem)    if (++(elem) >= CMD_Q_SIZE)\
    (elem) = 0;
```

```
typedef struct {
    int    cmd_elem;
    COMMAND    *cmd_q;
}    CMD_Q;
```

```
typedef struct {
    u_char    crc_err;
    u_char    parity_err;
    u_char    buf_ovr;
    u_char    buf_avail;
    u_char    pkt_drop;
    u_char    cell_drop;
}    HWNIU_STATS;
```

```
typedef struct {
    u_short    rx_packets;
```

//6
if_niu.h
-78-

```

u_short      tx_packets;
HWNIU_STATS  stats;
u_char       reserved[6];
} NIU_STATUS;

```

```

typedef struct {
    caddr_t    cmd_start; /* command q start */
    caddr_t    cmd_end; /* command q end */
    caddr_t    done_start; /* completed q start */
    caddr_t    done_end; /* completed q end */
    caddr_t    status_start; /* status location */
} HOST_BASE;

```

```

#define MAP_CMD_Q      0
#define MAP_DONE_Q     1
#define MAP_STATUS     2
#define MAP_BASE      3

```

```

typedef struct {
    caddr_t    base_dma; /* host base dma address */
    caddr_t    status_dma; /* status dma address */
    caddr_t    cmd_dma; /* cmd q dma address */
    caddr_t    done_dma; /* done q dma address */
} DMA_ADDR;

```

```

/* board ids */
#define NIU_REV2 0
#define NIU_REV3 1
#define PNIO_REV1 2
#define PNIO_REV2 3
#define PNIO_REV3 4
struct niu_dev {
    u_char      type;
    u_short     tag; /* tag for each command */
    CMD_Q       cmd_q; /* command q */
    CMD_Q       done_q; /* completed q */
    NIU_STATUS  status; /* hw_niu status location */
    HOST_BASE   base; /* host/niu io base structure */
    DMA_ADDR    dma_addr; /* mapped dma address */
    int         priority; /* interrupt priority */
    struct hw_niu_reg niu_reg; /* address of registers on
    * niu board */
    struct niu_stats stats; /* niu statistics */
    DMA_DESC    desc[CMD_Q_SIZE]; /* descriptor pool */
    int         intr_timeout; /* interrupt timeout counter */
    struct niu_addr_reg niu_addr; /* address of registers on
    * niu board */
    struct niu_value_reg niu_value; /* contents of registers on
    * niu board */
    struct dmac_addr_reg dmac_addr; /* address of registers on
    * 64853A SBus controller */
}

```

//7
if_niu.h
-79-

```

struct dmac_value_reg dmac_value; /* contents of registers
    * on L64853A SBUS
    * controller */
int (*sendpkt) (); /* hw specific routine to
    * send queued pkts */
int (*reset) (); /* hw specific routine to
    * reset hw */
#define NIU_TYPE_SW 1
#define NIU_TYPE_HW 2
u_short direction; /* receive or transmit
    * packets */
int board_id; /* niu board revision */
int intr_state; /* is in interrupt state */
int post_rxbuf; /* count of rx buffers to be
    * posted */
struct ifqueue sendq;
int macaddr[2];
};

extern struct niu_dev niu_info[];
extern int cell_flag; /* set for trasmission of raw 53 byte
    * cells */

#define NIU_IFUNIT_TO_IOUNIT(ifunit) (atm_glob->atmif[ifunit].ati_pcif->pc_num)

```

//8

lm.c
-80-

```

/* lm.c
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */
*****END*****/

#ifdef CERNEL

#include "ipc_def.h"
#include "net_def.h"
#include <global_def.h>
#include <driver.h>

#undef lm_init

#else /* ifndef CERNEL */
#include <stdint.h>
#include <global_def.h>
#include <ITC_if.h>
#include <driver.h>

#include <RT_if.h>
#include <timer.h>
#include <RT_def.h>
#include <enet_if.h>
#include <net_def.h>

#define ERRLOG printdbg
#define printf printdbg

#endif /* ifdef CERNEL */

#include "unipdu.h"
#include "nnipdus.h"
#include "altask_gl.h"
#include "sigtask_gl.h"
#include "svctask_gl.h"
#include "svc_if.h"
#include "snmp_incl.h"
#include "AAL_if.h"
#include "wdb_if.h"
#include "q.h"
#include "bits.h"
#include "lm.h"

lm_tcb_t    *lm_init();

```

119

lm.c

-81-

```

#ifdef CERNEL
#include <stdio.h>

main(argc, argv, environ)
    int      argc;
    char     *argv[];
    char     **environ[];
{
    tINT32    generic;
    tINT32    instance;
    tINT32    status;
    tUINT8    test_mode;

    generic = TID_LM;
    instance = 0;

    if ((status = SetTid(generic, instance)) != RT_SUCCESS) {
        printf("lm: SetTid Failed\n");
    } else {
        lm_main();
    }
}
#endif          /* ifdef CERNEL */

lm_main()
{
    lm_tcb_t  *tcb;
    struct TimerBlock *tmr_blk;
    tUINT32   *msg;
    tINT32     delay;
    tUINT32    timerid;
    tUINT32    timerarg;

    tcb = lm_init();
    if (tcb == NULL) {
        printf("lm: init failed");
        return;
    }
    tmr_blk = tcb->tmr_blk;
    timerid = tcb->timerid;
    timerarg = tcb->timerarg;

    while (TRUE) {
        delay = 0;
        while (delay <= 0) {
            delay = TimerCheck(tmr_blk, &timerid, &timerarg);
            if (delay <= 0) {
                lm_srvc_timer(2);
                RTC_TimerSet(tmr_blk, (GetTime() + (STGRAN)),
                    timerid, timerarg);
            }
        }
    }
}

```

1020

lm.c

-82-

```

    }
    }
    msg = (tUINT32 *) ReqMsg(LM_EX_MSK, delay);
    if (msg != NULL) {
        lm_srvc_msg(msg);
        free(msg);
    }
}

lm_srvc_timer(delay)
tUINT32      delay;
{
    SETUP_TCB;

}

lm_srvc_msg(msg)
tITC_HEADER  *msg;
{
    int      ret;
    int      lm_crt_cfg();
    SETUP_TCB;

    ret = RT_SUCCESS;
    printf("lm_srvc_msg, MsgType = %d\r\n", msg->MsgType);

    switch (msg->MsgType) {
    case TA_AAL_IND_RECEIVE:
        lm_srvc_aal_msg(msg);
        break;
    case U_DTIND:
        lm_srvc_svc_msg(msg);
        break;
    case SNMPA_MGMT_GET:
        lm_srvc_mgmt_get(msg);
        break;
    case SNMPA_MGMT_VALIDATE:
        lm_srvc_mgmt_validate(msg);
        break;
    case SNMPA_MGMT_COMMIT:
        lm_srvc_mgmt_commit(msg);
        break;
    case SNMPA_MGMT_GETNEXT:
        lm_srvc_mgmt_getnext(msg);
        break;
    case SNMPA_CHECKIN_MSG:
        SendProxyCheckin(MHW_GetCardType(),

```

121
lm.c
-83-

```

        MHW_GetSlotId());
    break;
default:
    if ((msg->MsgType >= MSG_WDB_BASE) &&
        (msg->MsgType <= MSG_WDB_TOP)) {
        wdb_process_msg(lm_crt_cfg, msg);
    } else {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    break;
}
return (ret);

err_exit:
return (ret);
}

lm_svc_aal_msg(msg)
    TAAL_TA_IND_RX *msg;
{
    int         ret;
    lm_alan_cfg_enq_t *aal_msg;
    tUINT32     pvc;
    tUINT32     vci;

    aal_msg = (lm_alan_cfg_enq_t *) msg->Rx.Buffer;
    ret = RT_SUCCESS;

    vci = ((lm_atm_hdr_t *) & (msg->RxATM_Hdr))->vci;
    pvc = VCI_TO_PVC(vci);
    if (aal_msg->lm_hdr.lh_pdu_type != NN_PDU_STATUS_ENQ ||
        aal_msg->lm_hdr.lh_proto != NNI_PROTOCOL ||
        pvc != NNI_NAC_VCI) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    switch (aal_msg->enq.elem_type) {
    case ALAN_CFG_ENQ:
        ret = lm_svc_alan_cfg_enq(msg);
        break;
    case LMI_CONFIG_ENQ:
        ret = lm_svc_es_cfg_enq(msg);
        break;
    default:
        goto err_exit;
        break;
    }
    return (ret);
}

```

122

lm.c

-84-

```

err_exit:
    return (ret);
}

lm_srvc_alan_cfg_enq(msg)
    tAAL_TA_IND_RX *msg;
{
    int         ret;
    lm_alan_cfg_enq_t *alan_enq;
    tALANCFG_ENQ *enq;
    tALANCFG_RESP *resp;
    tATMADDR     paddr;
    tUINT32      in_srv;
    tUINT32      in_srv_mask;
    int          i;
    int          max_port;
    lm_port_addr_t tst;
    lm_port_t    *port;
    lm_mac_t     *mac;
    qlink_t      *link;
    lm_mac_vlan_t *mv;
    tUINT32      tx_vci;
    tUINT32      rx_vci;
    tUINT8       rx_shelf;
    tUINT8       rx_slot;
    tUINT8       rx_port;
    lm_prefix_t  prefix;
    lm_atm_hdr_t atm_hdr;
    SETUP_TCB;

    ret = RT_SUCCESS;
    alan_enq = (lm_alan_cfg_enq_t *) msg->Rx.Buffer;
    enq = &alan_enq->enq;

    in_srv = 0;
    in_srv_mask = 0x80000000;
    max_port = MAX_PORTS_PER_SLOT > enq->num_ports ?
        enq->num_ports : MAX_PORTS_PER_SLOT;
    LM_INIT_PORT_ADDR(&tst, tcb->my_node, tcb->my_shelf, enq->slotid, 0);
    for (i = 0; i < max_port; i++) {
        tst.aa_lanum = 0;
        tst.aa_port = i;
        port = FIND_PORT(tcb->port_q, &tst);
        if (port != NULL) {
            mac = port->mac;
            if (IIS_EMPTY_Q(port->pv_q)) {
                paddr[i] = tst;
                in_srv |= in_srv_mask;
            }
        }
    }
}

```

/23

lm.c
-85-

```

    if (mac != NULL && !IS_EMPTY_Q(mac->mv_q)) {
        link = HEAD_Q(mac->mv_q);
        mv = (lm_mac_vlan_t *) link->data;
        tst.aa_lanum = mv->mld;
        paddr[i] = tst;
        ln_srvc |= ln_srvc_mask;
    }
    ln_srvc_mask >>= 1;
}

atm_hdr = *((lm_atm_hdr_t *) &msg->RxATM_Hdr);
prefix = *((lm_prefix_t *) &msg->RxPrefix);
rx_vci = atm_hdr.vci;
tx_vci = (rx_vci & (~SIG_PVCI_MASK)) | PVCI_TO_VCI(m(NNI_SIG_VCI));
rx_shelf = VCI_TO_SHELF(rx_vci);
rx_slot = VCI_TO_SLOT(rx_vci);
rx_port = VCI_TO_PORT(rx_vci);

BUILD_ATM_HDR(&atm_hdr, tx_vci);
BUILD_UCAST_PREFIX(&prefix, rx_shelf, rx_slot, rx_port);

lm_send_alan_cfg(prefix, atm_hdr, enq->slotid, ln_srvc, max_port, paddr);
return (ret);

err_exit:
return (ret);
}

lm_srvc_es_cfg_enq(msg)
    TAAL_TA_IND_RX *msg;
{
    int ret;
    lm_es_cfg_enq_t *es_enq;
    tCFG_ELEM *enq;
    lm_mac_t *mac;
    lm_port_t *port;
    tUINT32 rx_vci;
    tUINT8 rx_shelf;
    tUINT8 rx_slot;
    tUINT8 rx_port;
    tUINT32 tx_vci;
    lm_prefix_t prefix;
    lm_atm_hdr_t atm_hdr;
    lm_mac_addr_t *mac_addr;
    lm_port_addr_t port_addr;
    lm_es_cfg_resp_t *resp;
    int resp_len;
    int i;
    SETUP_TCB;

```

124
lm.c
-86-

```

ret = RT_SUCCESS;
es_enq = (lm_es_cfg_enq_t *) msg->Rx.Buffer;
enq = &es_enq->enq;
mac_addr = &enq->af_my_address;

atm_hdr = *((lm_atm_hdr_t *) &msg->RxATM_Hdr);
prefix = *((lm_prefix_t *) &msg->RxPrefix);
rx_vci = atm_hdr.vci;
tx_vci = (rx_vci & (~SIG_PVCI_MASK)) | PVCI_TO_VCI(m(NNI_SIG_VCI));
rx_shelf = VCI_TO_SHELF(rx_vci);
rx_slot = VCI_TO_SLOT(rx_vci);
rx_port = VCI_TO_PORT(rx_vci);
LM_INIT_PORT_ADDR(&port_addr, tcb->my_node, rx_shelf, rx_slot,
rx_port);

port = FIND_PORT(tcb->port_q, &port_addr);
mac = FIND_MAC(tcb->mac_q, mac_addr);
if (mac == NULL) {
    mac = add_mac(mac_addr);
    if (port != NULL) {
        atch_mac_port(mac, port);
        lm_dup_port_dflts(port, mac);
    }
}
if (port != NULL && (port->mac != mac || mac->port != port)) {
    free_mac_port(port->mac, port);
    atch_mac_port(mac, port);
}
resp = lm_build_es_cfg_resp(mac, enq, &resp_len);
if (resp == NULL) {
    ret = IRT_SUCCESS;
    goto err_exit;
}
BUILD_ATM_HDR(&atm_hdr, tx_vci);
BUILD_UCAST_PREFIX(&prefix, rx_shelf, rx_slot, rx_port);

ret = lm_send_es_cfg_resp(prefix, atm_hdr, resp, resp_len);
return (ret);

err_exit:
return (ret);
}

lm_svc_svc_msg(msg)
    tAALUSRMSG *msg;
{
    tLMIHDR *lmi_hdr;
    tSETUP *setup;

```

125

lm.c
-87-

```

    lmi_hdr = (tLMIHDR *) & msg->U_PDU;
    switch (lmi_hdr->lh_pdu_type) {
    case SDU_SETUP_IND:
        lm_svc_svc_setup_ind(msg);
        break;
    case SDU_SETUP_COMP:
        break;
    case SDU_RELEASE_IND:
        lm_svc_svc_rel_ind(msg);
        break;
    default:
        break;
    }
}

lm_svc_svc_rel_ind(msg)
    struct svcif *msg;
{
}

lm_svc_svc_setup_ind(msg)
    struct svcif *msg;
{
    struct svcif *resp;
    tLMIHDR      *lmi_hdr;
    tREL_REQ     *rel;
    int          resp_len;
    int          ret;
    lm_mac_t     *mac;
    lm_port_t    *port;
    lm_vlan_t     *vlan;
    lm_mac_vlan_t *mv;
    lm_mlid_t     mld;
    tSETUP       *rx_setup;
    tSETUP       *tx_setup;
    lm_port_addr_t port_addr;
    lm_vc_addr_t  vc_addr;
    lm_vc_t       *vc;
    tUINT8        *vpcl;
    SETUP_TCB;

    ret = RT_SUCCESS;

    resp_len = SVCIF_PDU_OFFSET + sizeof(*tx_setup) +
        sizeof(struct lmi_parm);
    resp = (struct svcif *) ReqMsgMemZero(resp_len);
    if (resp == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }

```

126

lm.c
-88-

```

    }
    rx_setup = (tSETUP *) & msg->lm_hdr;
    tx_setup = (tSETUP *) & resp->lm_hdr;
    *tx_setup = *rx_setup;
    lm_hdr = (tLMHDR *) & tx_setup->lm_hdr;
    lm_hdr->lh_pdu_type = SDU_SETUP_RESP;
    port_addr = rx_setup->lm_caller;
    mlid = port_addr.aa_lannum;
    port_addr.aa_lannum = 0;
    port = FIND_PORT(tcb->port_q, &port_addr);
    if (port == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    mac = port->mac;
    if (mac == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    mv = FIND_MLID(mac->mv_q, mlid);
    if (mv == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    vlan = mv->vlan;
    if (vlan == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    LM_INIT_VC_ADDR(&vc_addr, vlan->vlan_id, &rx_setup->lm_callee);
    vc = FIND_VC(vlan->vc_q, &vc_addr);
    if (vc == NULL) {
        vc = add_vc(&vc_addr);
    }
    if (vc == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    vc->ref_cnt++;
    vpci = (tUINT8 *) tx_setup + sizeof(*tx_setup);
    LMI_ADD_ELEMENT(vpci, LMI_OVPCI, vc->bid);

    lm_hdr->lh_cref_type |= LMI_CREFDIRECTION_MASK;
    ret = lm_send_svc_msg(resp, resp_len);

    return (ret);

```

```

err_exit:
    if (resp != NULL) {

```

127

lm.c

-89-

```
lmi_hdr->lh_cref_type |= LMI_CREFDIRECTION_MASK;  
lm_send_svc_rel_req(lmi_hdr, INVALID_STATE);  
free(resp);
```

```
}  
return (ret);
```

```
}
```

128
lm.h
-80-

```

/* lm.h
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */
*****END*****/

#ifndef LM_H
#define LM_H

#define LM_VB_QUIET (0)
#define LM_VB_ERRS (1)
#define LM_VB_TERSE (2)
#define LM_VB_VERBOSE (3)
#define LM_VB_MSGS (4)
#define LM_VB_ALL (999)

#define CHK_VB(level) (tcb->verbose >= level)

#define LM_MAX_VLAN_NAME (17)
#define LM_INDENT (2)
#define LM_DFLT_MTU_SIZE (9100)
#define LM_DFLT_NUM_MCASTS (4)
#define LM_MAX_MLID (256)
#define LM_MAX_BID (1024)
#define LM_MAX_MID (1024)
#define MAX_SLOTS (16)
#define MAX_PORTS_PER_SLOT (8)
#define MAX_PORTS (MAX_SLOTS * MAX_PORTS_PER_SLOT)
#define LM_AAL_EX (EX INDICATION)
#define LM_AAL_EX_MSK (M EX INDICATION)
#define LM_START_VCI (0x3000)
#define LM_END_VCI (0x3fff)
#define LM_INSTANCE (0)
#define LM_AAL_SID (MAKE_SSID(TID LM, LM_INSTANCE, LM_AAL_EX))
#define LM_EX_MSK (M EX INDICATION)
#define SIZE_MID_BITS (LM_MAX_MID / (8 * SIZE_BITS))
#define SIZE_MLID_BITS (LM_MAX_MLID / (8 * SIZE_BITS))
#define SIZE_BID_BITS (LM_MAX_BID / (8 * SIZE_BITS))
#define LM_CLR_MAC_ADDR(maddr) \
    ((maddr)->aa_long[0] = (maddr)->aa_long[1] = 0, (maddr)->aa_type = AAT_MAC)
#define LM_CLR_PORT_ADDR(paddr) \
    ((paddr)->aa_long[0] = (paddr)->aa_long[1] = 0, (paddr)->aa_type = AAT_PORT, \
    (paddr)->aa_country = USA)
#define LM_CLR_VC_ADDR(vcaddr) \
    ((vcaddr)->vlan_id = 0, LM_CLR_MAC_ADDR(&((vcaddr)->mac_addr)))
#define LM_INIT_PORT_ADDR(paddr, node, shelf, slot, port) \
    (LM_CLR_PORT_ADDR(paddr), \
    (paddr)->aa_node = node, (paddr)->aa_shelf = shelf, (paddr)->aa_slot = slot, \

```

129

lm.h

-91-

```

(paddr)->aa_port = port, (paddr)->aa_lanum = 0)
#define LM_INIT_MAC_ADDR(maddr, mac_addr)\
((maddr)->aa_long[0] = (mac_addr)->aa_long[0], \
(maddr)->aa_long[1] = (mac_addr)->aa_long[1], \
(maddr)->aa_type = AAT_MAC)
#define LM_INIT_VC_ADDR(vcaddr, vid, maddr)\
(LM_CLR_VC_ADDR(vcaddr), (vcaddr)->vlan_id = (vid), \
LM_INIT_MAC_ADDR(&((vcaddr)->mac_addr), maddr))
#define LM_NUM_ELEM(ary) (sizeof(ary) / sizeof((ary)[0]))
#define BUILD_ITCH(Hdr, len, generic, instance, exch, mtype, mytid) \
{Hdr.Length=len; \
Hdr.Dest.Label.Pid=0; \
Hdr.Dest.Label.Sid = \
MAKE_SSID(generic,instance,exch);\
Hdr.Dest.Net = LOCAL_NET; \
Hdr.Dest.Node = LOCAL_NODE; \
GetPid(&Hdr.Orig.Label.Pid); \
Hdr.Orig.Label.Sid = \
MAKE_SSID(mytid.Generic, 0, EX_INDICATION);\
Hdr.MsgType = mtype;}

```

```

typedef tUINT32 lm_mld_t;
typedef tUINT32 lm_bid_t;
typedef tUINT16 lm_vlan_id_t;

```

```

typedef struct lm_prefix_s {
    unsigned    pri:2;
    unsigned    tag_a:6;
    unsigned    fill1:2;
    unsigned    rp:1;
    unsigned    nrc:1;
    unsigned    cos:4;
    unsigned    fill2:1;
    unsigned    br:1;
    unsigned    vem:1;
    unsigned    mb:1;
    unsigned    tag_b:4;
    unsigned    fill3:2;
    unsigned    tag_c:6;
} lm_prefix_t;

```

```

#define SIZE_LM_PREFIX (sizeof(lm_prefix_t))

```

```

#define CLR_PREFIX(pfx) (((tUINT32*)(pfx)) = 0)

```

```

#define BUILD_UCAST_PREFIX(pfx, shelf, slot, port)\
(CLR_PREFIX(pfx), (pfx)->tag_a = shelf, (pfx)->tag_b = slot, \
(pfx)->tag_c = port, (pfx)->rp = 1)

```

```

/* Fix RPA 10 Mar 92 */

```

/20

lm.h

-92-

```

#define BUILD_MCAST_PREFIX(pfx, bid)\
  (CLR_PREFIX(pfx), (pfx)->br = 1, (pfx)->tag_a = (((bid) & 0xfc00) >> 10),\
  (pfx)->tag_b = (((bid) & 0x03c0) >> 6), (pfx)->tag_c = ((bid) & 0x003f))

typedef struct lm_atm_hdr_s {
  unsigned   gfc:4;
  unsigned   vpi:8;
  unsigned   vci:16;
  unsigned   pt:2;
  unsigned   rsvd:1;
  unsigned   clp:1;
} lm_atm_hdr_t;
#define SIZE_LM_ATM_HDR (sizeof(lm_atm_hdr_t))

#define CLR_ATM_HDR(hdr) (*(tUINT32*)(hdr)) = 0)
#define BUILD_ATM_HDR(hdr, the_vci)\
  (CLR_ATM_HDR(hdr), (hdr)->vci = the_vci)

typedef struct lm_alan_cfg_enq_s {
  struct lmi_hdr lmi_hdr;
  tALANCFG_ENQ   enq;
} lm_alan_cfg_enq_t;
#define SIZE_LM_ALAN_CFG_ENQ (sizeof(lm_alan_cfg_enq_t))

typedef struct lm_alan_cfg_resp_s {
  struct lmi_hdr lmi_hdr;
  tALANCFG_ENQ   enq;
  tALANCFG_RESP   resp;
} lm_alan_cfg_resp_t;
#define SIZE_LM_ALAN_CFG_RESP (sizeof(lm_alan_cfg_resp_t))

typedef struct lm_es_cfg_resp_s {
  struct lmi_hdr lmi_hdr;
  tCFGELEM       enq;
  tCFGELEM       resp;
  tPORT_CFGELEM  paddr[1];
} lm_es_cfg_resp_t;
#define SIZE_LM_ES_CFG_RESP (sizeof(lm_es_cfg_resp_t))

typedef struct lm_es_cfg_enq_s {
  struct lmi_hdr lmi_hdr;
  struct config_elem enq;
} lm_es_cfg_enq_t;
#define SIZE_LM_ES_CFG_ENQ (sizeof(lm_es_cfg_enq_t))

typedef struct atm_addr lm_mac_addr_t;
#define SIZE_LM_MAC_ADDR (sizeof(lm_mac_addr_t))

typedef struct atm_addr lm_port_addr_t;
#define SIZE_LM_PORT_ADDR (sizeof(lm_port_addr_t))

```

131

lm.h

-93-

```

typedef struct lm_vc_addr_s {
    lm_vlan_id_t   _vlan_id;
    struct atm_addr mac_addr;
} lm_vc_addr_t;
#define SIZE_LM_VC_ADDR (sizeof(lm_vc_addr_t))

#define aa_country    aa_u.aaw.aasw_nibble2
#define aa_shelf      aa_u.aaw.aasw_nibble3
#define aa_slot       aa_u.aaw.aasw_nibble4
#define aa_port       aa_u.aaw.aasw_nibble5

typedef struct lm_tcb_s {
    tTID            mytid;
    tPID            mypid;
    struct TimerBlock *tmr_blk;
    tUINT32         timerid;
    tUINT32         timerarg;
    tATMADDR        nac_atm_addr;
    tUINT32         nac_id;
    tAAL_KEY        my_aal_key;
    tUINT32         my_node;
    tUINT32         my_shelf;
    tUINT32         my_slot;
    lm_port_addr_t  port_tmplt;
    tUINT32         _cur_bid;
    bits_t          bid_bits[SIZE_BID_BITS];
    bits_t          mliid_bits[SIZE_MLIID_BITS];
    tUINT32         dflt_mtu_size;
    tUINT32         dflt_num_mcasts;
    tUINT32         verbose;
    tUINT32         do_cfg_wrts;
    char            vc_addr_buf[200];
    char            mac_addr_buf[sizeof(lm_mac_addr_t) * 3 + 1];
    char            port_addr_buf[40];
    char            vlan_id_buf[10];
    queue_t         port_queue;
    queue_t         vlan_queue;
    queue_t         mac_queue;
    queue_t         mv_queue;
    queue_t         pv_queue;
    queue_t         vc_queue;
    queue_t         *port_q;
    queue_t         *vlan_q;
    queue_t         *mac_q;
    queue_t         *mv_q;
    queue_t         *pv_q;
    queue_t         *vc_q;
} lm_tcb_t;
#define SIZE_LM_TCB (sizeof(lm_tcb_t))

```

/32

lm.h

-94-

```

typedef struct lm_mac_s {
    qlink_t    mac_link;
    queue_t    mv_queue;
    queue_t    *mv_q;
    struct lm_port_s *port;
    lm_mac_addr_t mac_addr;
    bits_t     mld_bits[SIZE_MLID_BITS];
} lm_mac_t;
#define SIZE_LM_MAC (sizeof(lm_mac_t))

typedef struct lm_vlan_s {
    qlink_t    vlan_link;
    queue_t    pv_queue;
    queue_t    *pv_q;
    queue_t    mv_queue;
    queue_t    *mv_q;
    queue_t    vc_queue;
    queue_t    *vc_q;
    queue_t    free_vc_queue;
    queue_t    *free_vc_q;
    lm_vlan_id_t vlan_id;
    tUINT32    mtu_size;
    tUINT32    num_mcasts;
    lm_mld_t    dflt_mld;
    char       vlan_name[LM_MAX_VLAN_NAME];
    bits_t     mid_bits[SIZE_MID_BITS];
} lm_vlan_t;
#define SIZE_LM_VLAN (sizeof(lm_vlan_t))

typedef struct lm_port_s {
    qlink_t    port_link;
    lm_port_addr_t port_addr;
    lm_mac_t    *mac;
    queue_t    pv_queue;
    queue_t    *pv_q;
    bits_t     mld_bits[SIZE_MLID_BITS];
} lm_port_t;
#define SIZE_LM_PORT (sizeof(lm_port_t))

typedef struct lm_vc_s {
    qlink_t    vc_link;
    qlink_t    vlan_link;
    lm_vlan_t    *vlan;
    lm_vc_addr_t vc_addr;
    lm_bld_t    bld;
    tUINT32    ref_cnt;
} lm_vc_t;
#define SIZE_LM_VC (sizeof(lm_vc_t))

```

133
lm.h
-95-

```
typedef struct lm_port_vlan_s {
    qlink_t      pv_link;
    qlink_t      port_link;
    qlink_t      vlan_link;
    lm_vlan_t     *vlan;
    lm_port_t     *port;
    lm_mlid_t     mlid;
    lm_vlan_id_t  vlan_id;
    lm_port_addr_t port_addr;
} lm_port_vlan_t;
#define SIZE_LM_PORT_VLAN (sizeof(lm_port_vlan_t))
```

```
typedef struct lm_mac_vlan_s {
    qlink_t      mv_link;
    qlink_t      mac_link;
    qlink_t      vlan_link;
    lm_mac_t     *mac;
    lm_vlan_t     *vlan;
    lm_vlan_id_t vlan_id;
    lm_mac_addr_t mac_addr;
    tUINT16      mlid;
    lm_mlid_t     mlid;
} lm_mac_vlan_t;
#define SIZE_LM_MAC_VLAN (sizeof(lm_mac_vlan_t))
```

```
typedef struct lm_cfg_glbl_s {
    tUINT32      dflt_mtu_size;
    tUINT32      dflt_num_mcasts;
} lm_cfg_glbl_t;
#define SIZE_LM_CFG_GLBL (sizeof(lm_cfg_glbl_t))
```

```
typedef struct lm_cfg_vlan_s {
    tUINT32      dflt_mlid;
    tUINT32      num_mcasts;
    tUINT32      mtu_size;
    char         vlan_name[LM_MAX_VLAN_NAME];
} lm_cfg_vlan_t;
#define SIZE_LM_CFG_VLAN (sizeof(lm_cfg_vlan_t))
```

```
typedef struct lm_cfg_port_s {
    tUINT32      el_tonto;
} lm_cfg_port_t;
#define SIZE_LM_CFG_PORT (sizeof(lm_cfg_port_t))
```

```
typedef struct lm_cfg_mac_s {
    tUINT32      el_tonto;
} lm_cfg_mac_t;
#define SIZE_LM_CFG_MAC (sizeof(lm_cfg_mac_t))
```

```
typedef struct lm_cfg_pv_s {
```

134
lm.h
-86-

```

    tUINT32      mld;
}
    lm_cfg_pv t;
#define SIZE_LM_CFG_PV      (sizeof(lm_cfg_pv_t))

typedef struct lm_cfg_mv_s {
    tUINT32      mld;
}
    lm_cfg_mv t;
#define SIZE_LM_CFG_MV      (sizeof(lm_cfg_mv_t))

typedef struct lm_cfg_vc_s {
    tUINT32      el_tonto;
}
    lm_cfg_vc t;
#define SIZE_LM_CFG_VC      (sizeof(lm_cfg_vc_t))

typedef struct lm_glbl_cfg_key_s {
    tUINT32      tag;
}
    lm_glbl_cfg_key t;
#define SIZE_LM_GLBL_CFG_KEY      (sizeof(lm_glbl_cfg_key_t))

typedef struct lm_vlan_cfg_key_s {
    tUINT32      tag;
    lm_vlan_id t   vlan_id;
}
    lm_vlan_cfg_key t;
#define SIZE_LM_VLAN_CFG_KEY      (sizeof(lm_vlan_cfg_key_t))

typedef struct lm_port_cfg_key_s {
    tUINT32      tag;
    lm_port_addr t port_addr;
}
    lm_port_cfg_key t;
#define SIZE_LM_PORT_CFG_KEY      (sizeof(lm_port_cfg_key_t))

typedef struct lm_mac_cfg_key_s {
    tUINT32      tag;
    lm_mac_addr t mac_addr;
}
    lm_mac_cfg_key t;
#define SIZE_LM_MAC_CFG_KEY      (sizeof(lm_mac_cfg_key_t))

typedef struct lm_vc_cfg_key_s {
    tUINT32      tag;
    lm_vc_addr t   vc_addr;
}
    lm_vc_cfg_key t;
#define SIZE_LM_VC_CFG_KEY      (sizeof(lm_vc_cfg_key_t))

typedef struct lm_pv_cfg_key_s {
    tUINT32      tag;
    lm_port_addr t port_addr;
    lm_vlan_id t   vlan_id;
}
    lm_pv_cfg_key t;
#define SIZE_LM_PV_CFG_KEY      (sizeof(lm_pv_cfg_key_t))

```

/35

lm.h

-97-

```

typedef struct lm_mv_cfg_key_s {
    tUINT32      tag;
    lm_mac_addr_t mac_addr;
    lm_vlan_id_t vlan_id;
} lm_mv_cfg_key_t;
#define SIZE_LM_MV_CFG_KEY (sizeof(lm_mv_cfg_key_t))

typedef union cfg_key_u {
    tUINT32      tag;
    lm_glbl_cfg_key_t glbl_key;
    lm_vlan_cfg_key_t vlan_key;
    lm_port_cfg_key_t port_key;
    lm_mac_cfg_key_t mac_key;
    lm_vc_cfg_key_t vc_key;
    lm_pv_cfg_key_t pv_key;
    lm_mv_cfg_key_t mv_key;
} lm_cfg_key_t;
#define SIZE_LM_CFG_KEY (sizeof(lm_cfg_key_t))

#define NULL_CFG_KEY (0)
#define GLBL_CFG_KEY (1)
#define VLAN_CFG_KEY (2)
#define PORT_CFG_KEY (3)
#define MAC_CFG_KEY (4)
#define VC_CFG_KEY (5)
#define PV_CFG_KEY (6)
#define MV_CFG_KEY (7)

#ifdef UNIX
extern tINT8 *GlobalP;
#undef printf
#endif

#define malloc(size) GetMem(size)
#define free(ptr) FreeMem(ptr)

#define FREE_Q(q, proc) ((int)traverse q(q, proc, NULL))
#define FREE_MV_Q(q) FREE_Q(q, free_mv)
#define FREE_PV_Q(q) FREE_Q(q, free_pv)
#define FREE_MAC_Q(q) FREE_Q(q, free_mac)
#define FREE_VLAN_Q(q) FREE_Q(q, free_vlan)
#define FREE_PORT_Q(q) FREE_Q(q, free_port)
#define FREE_VC_Q(q) FREE_Q(q, free_vc)
#define FREE_PORT_VLAN_Q(q) FREE_Q(q, free_port_vlan)
#define FREE_MAC_VLAN_Q(q) FREE_Q(q, free_mac_vlan)
#define FREE_MAC_MV_Q(q) FREE_Q(q, free_mac_mv)
#define FREE_VLAN_MV_Q(q) FREE_Q(q, free_vlan_mv)
#define FREE_PORT_PV_Q(q) FREE_Q(q, free_port_pv)
#define FREE_VLAN_PV_Q(q) FREE_Q(q, free_vlan_pv)

```

136

lm.h

-98-

```

#define ATCH_MAC_MV_Q(q, mac) ((int)traverse_q(q, atch_mac_mv, mac))
#define ATCH_VLAN_MV_Q(q, vlan) ((int)traverse_q(q, atch_vlan_mv, vlan))
#define ATCH_PORT_PV_Q(q, port) ((int)traverse_q(q, atch_port_pv, port))
#define ATCH_VLAN_PV_Q(q, vlan) ((int)traverse_q(q, atch_vlan_pv, vlan))

#define ATCH_MV_MAC_Q(q, mv) ((int)traverse_q(q, atch_mv_mac, mv))
#define ATCH_MV_VLAN_Q(q, mv) ((int)traverse_q(q, atch_mv_vlan, mv))
#define ATCH_PV_PORT_Q(q, pv) ((int)traverse_q(q, atch_pv_port, pv))
#define ATCH_PV_VLAN_Q(q, pv) ((int)traverse_q(q, atch_pv_vlan, pv))

#define FIND_MAC(q, mac) ((lm_mac_t*)traverse_q(q, cmp_mac, mac))
#define FIND_PORT(q, port) ((lm_port_t*)traverse_q(q, cmp_port, port))
#define FIND_VLAN(q, vlan) ((lm_vlan_t*)traverse_q(q, cmp_vlan, vlan))
#define FIND_VC(q, vc) ((lm_vc_t*)traverse_q(q, cmp_vc, vc))
#define FIND_MV(q, mv) ((lm_mac_vlan_t*)traverse_q(q, cmp_mv, mv))
#define FIND_PV(q, pv) ((lm_port_vlan_t*)traverse_q(q, cmp_pv, pv))
#define FIND_MLID(q, mlid) ((lm_mac_vlan_t*)traverse_q(q, cmp_mlid, mlid))

#define FINDNEXT_MAC(q, mac) ((lm_mac_t*)traverse_q(q, cmpnext_mac, mac))
#define FINDNEXT_PORT(q, port) ((lm_port_t*)traverse_q(q, cmpnext_port, port))
#define FINDNEXT_VLAN(q, vlan) ((lm_vlan_t*)traverse_q(q, cmpnext_vlan, vlan))
#define FINDNEXT_VC(q, vc) ((lm_vc_t*)traverse_q(q, cmpnext_vc, vc))
#define FINDNEXT_MV(q, mv) ((lm_mac_vlan_t*)traverse_q(q, cmpnext_mv, mv))
#define FINDNEXT_PV(q, pv) ((lm_port_vlan_t*)traverse_q(q, cmpnext_pv, pv))
#define FINDNEXT_MLID(q, mlid) ((lm_mac_vlan_t*)traverse_q(q, cmpnext_mlid, mlid))

#define PUTQ_SORTED_MAC(link, q) (putq_sorted(link, q, qpsc_mac))
#define PUTQ_SORTED_PORT(link, q) (putq_sorted(link, q, qpsc_port))
#define PUTQ_SORTED_VLAN(link, q) (putq_sorted(link, q, qpsc_vlan))
#define PUTQ_SORTED_VC(link, q) (putq_sorted(link, q, qpsc_vc))
#define PUTQ_SORTED_PV(link, q) (putq_sorted(link, q, qpsc_pv))
#define PUTQ_SORTED_MV(link, q) (putq_sorted(link, q, qpsc_mv))

#define PRINT_Q(q, proc, indent) ((int)traverse_q(q, proc, indent))
#define PRINT_VLAN_Q(q, indent) PRINT_Q(q, print_vlan, indent)
#define PRINT_MAC_Q(q, indent) PRINT_Q(q, print_mac, indent)
#define PRINT_PORT_Q(q, indent) PRINT_Q(q, print_port, indent)
#define PRINT_PV_Q(q, indent) PRINT_Q(q, print_pv, indent)
#define PRINT_MV_Q(q, indent) PRINT_Q(q, print_mv, indent)
#define PRINT_VC_Q(q, indent) PRINT_Q(q, print_vc, indent)

#define pGlobalData ((lm_tcb_t*)GlobalP)

#define DEFINE_TCB lm_tcb_t *tcb
#define SETUP_TCB DEFINE_TCB = pGlobalData

extern struct AgentMsg *lm_cp_mgmt_msg0;
extern lm_es_cfg_resp_t *lm_build_es_cfg_resp0;
extern lm_mac_t *add_mac0;
extern lm_port_t *add_port0;

```

137

lm.h

-99-

```
extern lm_vlan_t *add_vlan();
extern lm_vc_t *add_vc();
extern lm_vc_t *get_free_vc();
extern lm_vc_t *add_free_vc();
extern lm_mac_vlan_t *add_mv();
extern lm_port_vlan_t *add_pv();
extern lm_mac_t *cmp_mac();
extern lm_port_t *cmp_port();
extern lm_vlan_t *cmp_vlan();
extern lm_vc_t *cmp_vc();
extern lm_port_vlan_t *cmp_pv();
extern lm_mac_vlan_t *cmp_mv();
extern lm_mac_vlan_t *cmp_mild();
extern lm_mac_t *cmpnext_mac();
extern lm_port_t *cmpnext_port();
extern lm_vlan_t *cmpnext_vlan();
extern lm_vc_t *cmpnext_vc();
extern lm_port_vlan_t *cmpnext_pv();
extern lm_mac_vlan_t *cmpnext_mv();
extern lm_mac_vlan_t *cmpnext_mild();
extern int qpsc_mac();
extern int qpsc_port();
extern int qpsc_vlan();
extern int qpsc_vc();
extern int qpsc_pv();
extern int qpsc_mv();
extern int atch_mac_mv();
extern int atch_vlan_mv();
extern int atch_port_pv();
extern int atch_vlan_pv();
extern int atch_pv_vlan();
extern int atch_pv_port();
extern int atch_mv_mac();
extern int atch_mv_vlan();
extern int free_mac();
extern int free_vlan();
extern int free_port();
extern int free_vc();
extern int free_port_vlan();
extern int free_mac_vlan();
extern int free_mac_mv();
extern int free_vlan_mv();
extern int free_port_pv();
extern int free_vlan_pv();
extern int free_mv();
extern int free_pv();
extern int print_mac();
extern int print_vlan();
extern int print_port();
extern int print_vc();
```

138
lm.h
-100-

```
extern int    print_mv();
extern int    print_pv();
extern char   *sprint_mac_addr();
extern char   *sprint_port_addr();
extern char   *sprint_vlan_id();
extern char   *sprint_vc_addr();

extern struct TimerBlock *TimerInit();

#endif          /* ifndef LM_H */
```

139
lm_cfg.c
-101-

```
/* lm_cfg.c
 *
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */
```

```
#ifdef CERNEL
```

```
#include "ipc_def.h"
#include "net_def.h"
#include <global_def.h>
#include <driver.h>
```

```
#undef lm_init
```

```
#else /* ifndef CERNEL */
```

```
#include <stdint.h>
#include <global_def.h>
#include <ITC_if.h>
#include <driver.h>
#include <RT_if.h>
#include <timer.h>
#include <RT_def.h>
#include <enet_if.h>
#include <net_def.h>
```

```
#define ERRLOG printdbg
#define printf printdbg
```

```
#endif /* ifdef CERNEL */
```

```
#include "unipdu.h"
#include "nnipdus.h"
#include "altask_gl.h"
#include "sigtask_gl.h"
#include "svctask_gl.h"
#include "svc_if.h"
#include "snmp_incl.h"
#include "AAL_if.h"
#include "wdb_if.h"
#include "q.h"
#include "bits.h"
#include "lm.h"
```

```
#define WDB_VERS (1)
#define WDB_PRI (1)
```

140
lm_cfg.c
-102-

```

lm_chg_glbl_cfg(dflt_mtu_size, dflt_num_mcasts, nac_addr, bid_bits, mlid_bits)
tUINT32      *dflt_mtu_size;
tUINT32      *dflt_num_mcasts;
tATMADDR     *nac_addr;
bits_t       *bid_bits;
bits_t       *mlid_bits;
{
    SETUP_TCB;

    if (dflt_mtu_size != NULL) {
        tcb->dflt_mtu_size = *dflt_mtu_size;
    }
    if (dflt_num_mcasts != NULL) {
        tcb->dflt_num_mcasts = *dflt_num_mcasts;
    }
    if (nac_addr != NULL) {
        tcb->nac_atm_addr = *nac_addr;
    }
    if (bid_bits != NULL) {
        bcopy(bid_bits, tcb->bid_bits, sizeof(tcb->bid_bits));
    }
    if (mlid_bits != NULL) {
        bcopy(mlid_bits, tcb->mlid_bits, sizeof(tcb->mlid_bits));
    }
    lm_wrt_glbl_cfg0;
}

lm_chg_vlan_cfg(vlan, dflt_mlid, num_mcasts, mtu_size, vlan_name)
lm_vlan_t     *vlan;
tUINT32      *dflt_mlid;
tUINT32      *num_mcasts;
tUINT32      *mtu_size;
tUINT32      *vlan_name;
{
    qlink_t     *link;
    lm_mac_t     *mac;
    lm_mac_vlan_t *mv;
    SETUP_TCB;

    if (dflt_mlid != NULL) {
        if (*dflt_mlid >= LM_MAX_MLID ||
            !bits_test_bit(*dflt_mlid, tcb->mlid_bits, SIZE_MLID_BITS)) {
            if (vlan->dflt_mlid < LM_MAX_MLID) {
                bits_free_bit(vlan->dflt_mlid, tcb->mlid_bits,
                    SIZE_MLID_BITS);
            }
            if (*dflt_mlid < LM_MAX_MLID) {
                bits_alloc_bit(*dflt_mlid, tcb->mlid_bits,
                    SIZE_MLID_BITS);
            }
        }
    }
}

```

141
lm_cfg.c
-103-

```

        vlan->dflt_mlid = *dflt_mlid;
    }
}
if (num_mcasts != NULL) {
    chg_num_vcs(vlan, *num_mcasts);
}
if (vlan_name != NULL) {
    int name_len;

    name_len = strlen(vlan_name);
    name_len = (name_len < LM_MAX_VLAN_NAME) ? name_len :
        LM_MAX_VLAN_NAME - 1;
    bcopy(vlan_name, vlan->vlan_name, name_len);
    vlan->vlan_name[name_len] = 0;
}
if (mtu_size != NULL) {
    vlan->mtu_size = *mtu_size;
}
lm_wrt_vlan_cfg(vlan);
for (link = HEAD_Q(vlan->mv_q); link != NULL;
     link = link->next) {
    mv = (lm_mac_vlan_t *) link->data;
    mac = mv->mac;
    if (mac != NULL) {
        lm_send_es_cfg_ind(mac);
    }
}
}

lm_chg_port_cfg(port, mac_addr, mlid_bits)
lm_port_t *port;
lm_mac_addr_t *mac_addr;
bits_t *mlid_bits;
{
    lm_mac_t *mac;
    SETUP_TCB;

    if (mac_addr != NULL) {
        if (port->mac != NULL) {
            mac = port->mac;
            free_mac(mac);
        }
        mac = add_mac(mac_addr);
        atch_mac_port(mac, port);
        lm_dup_port_dflts(port, mac);
    }
    if (mlid_bits != NULL) {
        bcopy(mlid_bits, port->mlid_bits, sizeof(port->mlid_bits));
    }
    lm_wrt_port_cfg(port);
}

```

142
lm_cfg.c
-104-

```

}

lm_chg_mac_cfg(mac, mlid_bits)
    lm_mac_t *mac;
    bits_t *mlid_bits;
{
    SETUP_TCB;

    if (mlid_bits != NULL) {
        bcopy(mlid_bits, mac->mlid_bits, sizeof(mlid_bits));
    }
    lm_wrt_mac_cfg(mac);
}

lm_chg_vc_cfg(vc, bid, ref_cnt)
    lm_vc_t *vc;
    lm_bid_t *bid;
    tUINT32 *ref_cnt;
{
    SETUP_TCB;

    if (bid != NULL) {
        if (!bits_tst_bit(*bid, tcb->bid_bits, SIZE_BID_BITS)) {
            bits_free_bit(vc->bid, tcb->bid_bits, SIZE_BID_BITS);
            bits_alloc_bit(*bid, tcb->bid_bits, SIZE_BID_BITS);
            vc->bid = *bid;
        }
    }

    if (ref_cnt != NULL) {
        vc->ref_cnt = *ref_cnt;
    }
    lm_wrt_vc_cfg(vc);
}

lm_chg_pv_cfg(pv, mlid)
    lm_port_vlan_t *pv;
    tUINT32 *mlid;
{
    lm_port_t *port;
    SETUP_TCB;

    port = pv->port;
    if (mlid != NULL && port != NULL) {
        if (!bits_tst_bit(*mlid, port->mlid_bits, SIZE_MLID_BITS)) {
            bits_free_bit(pv->mlid, port->mlid_bits, SIZE_MLID_BITS);
            bits_alloc_bit(*mlid, port->mlid_bits, SIZE_MLID_BITS);
            pv->mlid = *mlid;
        }
    }
}

```

143
lm_cfg.c
-105-

```

    lm_wrt_pv_cfg(pv);
}

lm_chg_mv_cfg(mv, mlid)
    lm_mac_vlan_t *mv;
    tUINT32 *mlid;
{
    lm_mac_t *mac;
    SETUP_TCB;

    mac = mv->mac;
    if (mlid != NULL && mac != NULL) {
        if (!bits_tst_bit(*mlid, mac->mlid_bits, SIZE_MLID_BITS)) {
            bits_free_bit(mv->mlid, mac->mlid_bits, SIZE_MLID_BITS);
            bits_alloc_bit(*mlid, mac->mlid_bits, SIZE_MLID_BITS);
            mv->mlid = *mlid;
        }
    }
    lm_wrt_mv_cfg(mv);
    if (mac != NULL) {
        lm_send_es_cfg_ind(mac);
    }
}

lm_rm_glbl_cfg()
{
    lm_glbl_cfg_key_t key;
    lm_cfg_glbl_t cfg;
    SETUP_TCB;

    key.tag = GLBL_CFG_KEY;

    wdb_send_remove_noack(&key, sizeof(key));
}

lm_rm_vlan_cfg(vlan)
    lm_vlan_t *vlan;
{
    lm_vlan_cfg_key_t key;
    lm_cfg_vlan_t cfg;
    SETUP_TCB;

    key.tag = VLAN_CFG_KEY;
    key.vlan_id = vlan->vlan_id;

    wdb_send_remove_noack(&key, sizeof(key));
}

lm_rm_port_cfg(port)
    lm_port_t *port;

```

144
lm_cfg.c
-108-

```

{
    lm_port_cfg_key_t key;
    lm_cfg_port_t cfg;
    SETUP_TCB;

    key.tag = PORT_CFG_KEY;
    key.port_addr = port->port_addr;

    wdb_send_remove_noack(&key, sizeof(key));
}

lm_rm_mac_cfg(mac)
    lm_mac_t *mac;
{
    lm_mac_cfg_key_t key;
    lm_cfg_mac_t cfg;
    SETUP_TCB;

    key.tag = MAC_CFG_KEY;
    key.mac_addr = mac->mac_addr;

    wdb_send_remove_noack(&key, sizeof(key));
}

lm_rm_vc_cfg(vc)
    lm_vc_t *vc;
{
    lm_vc_cfg_key_t key;
    lm_cfg_vc_t cfg;
    SETUP_TCB;

    key.tag = VC_CFG_KEY;
    key.vc_addr = vc->vc_addr;

    wdb_send_remove_noack(&key, sizeof(key));
}

lm_rm_pv_cfg(pv)
    lm_port_vlan_t *pv;
{
    lm_pv_cfg_key_t key;
    lm_cfg_pv_t cfg;
    SETUP_TCB;

    key.tag = PV_CFG_KEY;
    key.port_addr = pv->port_addr;
    key.vlan_id = pv->vlan_id;

    wdb_send_remove_noack(&key, sizeof(key));
}

```

145
lm_cfg.c
-107-

```

lm_rm_mv_cfg(mv)
    lm_mac_vlan_t *mv;
{
    lm_mv_cfg_key_t key;
    lm_cfg_mv_t cfg;
    SETUP_TCB;

    key.tag = MV_CFG_KEY;
    key.mac_addr = mv->mac_addr;
    key.vlan_id = mv->vlan_id;

    wdb_send_remove_noack(&key, sizeof(key));
}

lm_wrt_glbl_cfg()
{
    lm_glbl_cfg_key_t key;
    lm_cfg_glbl_t cfg;
    SETUP_TCB;

    if (tcb->do_cfg_wrts) {
        key.tag = GLBL_CFG_KEY;

        cfg.dflt_mtu_size = tcb->dflt_mtu_size;
        cfg.dflt_num_mcasts = tcb->dflt_num_mcasts;
        wdb_send_store_noack(&key, sizeof(key), &cfg, sizeof(cfg),
            WDB_VERS, WDB_PRI);
    }
}

lm_wrt_vlan_cfg(vlan)
    lm_vlan_t *vlan;
{
    lm_vlan_cfg_key_t key;
    lm_cfg_vlan_t cfg;
    SETUP_TCB;

    if (tcb->do_cfg_wrts) {
        key.tag = VLAN_CFG_KEY;
        key.vlan_id = vlan->vlan_id;

        cfg.dflt_mlid = vlan->dflt_mlid;
        cfg.num_mcasts = vlan->num_mcasts;
        cfg.mtu_size = vlan->mtu_size;
        bcopy(vlan->vlan_name, cfg.vlan_name, sizeof(cfg.vlan_name));
        wdb_send_store_noack(&key, sizeof(key), &cfg, sizeof(cfg),
            WDB_VERS, WDB_PRI);
    }
}

```

146
lm_cfg.c
-108-

```

lm_wrt_port_cfg(port)
{
    lm_port_t *port;
    lm_port_cfg_key_t key;
    lm_cfg_port_t cfg;
    SETUP_TCB;

    if (tcb->do_cfg_wrts) {
        key.tag = PORT_CFG_KEY;
        key.port_addr = port->port_addr;

        cfg.el_tonto = 0;
        wdb_send_store_noack(&key, sizeof(key), &cfg, sizeof(cfg),
            WDB_VERS, WDB_PRI);
    }
}

lm_wrt_mac_cfg(mac)
{
    lm_mac_t *mac;
    lm_mac_cfg_key_t key;
    lm_cfg_mac_t cfg;
    SETUP_TCB;

    if (tcb->do_cfg_wrts) {
        key.tag = MAC_CFG_KEY;
        key.mac_addr = mac->mac_addr;

        cfg.el_tonto = 0;
        wdb_send_store_noack(&key, sizeof(key), &cfg, sizeof(cfg),
            WDB_VERS, WDB_PRI);
    }
}

lm_wrt_vc_cfg(vc)
{
    lm_vc_t *vc;
    lm_vc_cfg_key_t key;
    lm_cfg_vc_t cfg;
    SETUP_TCB;

    if (tcb->do_cfg_wrts) {
        key.tag = VC_CFG_KEY;
        key.vc_addr = vc->vc_addr;

        cfg.el_tonto = 0;
        wdb_send_store_noack(&key, sizeof(key), &cfg, sizeof(cfg),
            WDB_VERS, WDB_PRI);
    }
}

```

147
lm_cfg.c
-109-

```

}

lm_wrt_pv_cfg(pv)
    lm_port_vlan_t *pv;
{
    lm_pv_cfg_key_t key;
    lm_cfg_pv_t    cfg;
    SETUP_TCB;

    if (tcb->do_cfg_wrts) {
        key.tag = PV_CFG_KEY;
        key.port_addr = pv->port_addr;
        key.vlan_id = pv->vlan_id;

        cfg.mlid = pv->mlid;
        wdb_send_store_noack(&key, sizeof(key), &cfg, sizeof(cfg),
                           WDB_VERS, WDB_PRI);
    }
}

lm_wrt_mv_cfg(mv)
    lm_mac_vlan_t *mv;
{
    lm_mv_cfg_key_t key;
    lm_cfg_mv_t    cfg;
    SETUP_TCB;

    if (tcb->do_cfg_wrts) {
        key.tag = MV_CFG_KEY;
        key.mac_addr = mv->mac_addr;
        key.vlan_id = mv->vlan_id;

        cfg.mlid = mv->mlid;
        wdb_send_store_noack(&key, sizeof(key), &cfg, sizeof(cfg),
                           WDB_VERS, WDB_PRI);
    }
}

lm_crt_cfg(key, klen, cfg, clen)
    lm_cfg_key_t *key;
    int          klen;
    tUINT8       *cfg;
    int          clen;
{
    switch (key->tag) {
        case NULL_CFG_KEY:
            break;
        case GLBL_CFG_KEY:
            lm_crt_glbl_cfg(key, cfg);
            break;
    }
}

```

148
lm_cfg.c
-110-

```

    case VLAN_CFG_KEY:
        lm_crt_vlan_cfg(key, cfg);
        break;
    case PORT_CFG_KEY:
        lm_crt_port_cfg(key, cfg);
        break;
    case MAC_CFG_KEY:
        lm_crt_mac_cfg(key, cfg);
        break;
    case VC_CFG_KEY:
        lm_crt_vc_cfg(key, cfg);
        break;
    case PV_CFG_KEY:
        lm_crt_pv_cfg(key, cfg);
        break;
    case MV_CFG_KEY:
        lm_crt_mv_cfg(key, cfg);
        break;
    default:
        break;
}
}

lm_crt_glbl_cfg(key, cfg)
    lm_glbl_cfg_key_t *key;
    lm_cfg_glbl_t *cfg;
{
    SETUP_TCB;

    lm_chg_glbl_cfg(&cfg->dflt_mtu_size, &cfg->dflt_num_mcasts, NULL, NULL,
        NULL);
}

lm_crt_vlan_cfg(key, cfg)
    lm_vlan_cfg_key_t *key;
    lm_cfg_vlan_t *cfg;
{
    lm_vlan_t *vlan;
    qlink_t *link;
    lm_mac_vlan_t *mv;
    lm_mac_t *mac;
    SETUP_TCB;

    vlan = FIND_VLAN(tcb->vlan_q, key->vlan_id);
    if (vlan == NULL) {
        vlan = add_vlan(key->vlan_id, cfg->mtu_size, cfg->num_mcasts,
            cfg->vlan_name);
    }
    lm_chg_vlan_cfg(vlan, &cfg->dflt_mld, &cfg->num_mcasts,
        &cfg->mtu_size, &cfg->vlan_name);
}

```

149

lm_cfg.c
-111-

```
}

lm crt_port_cfg(key, cfg)
    lm_port_cfg_key_t *key;
    lm_cfg_port_t *cfg;
{
    lm_port_t *port;
    SETUP_TCB;

    port = FIND_PORT(tcb->port_q, &key->port_addr);
    if (port == NULL) {
        port = add_port(&key->port_addr);
    }
}

lm crt_mac_cfg(key, cfg)
    lm_mac_cfg_key_t *key;
    lm_cfg_mac_t *cfg;
{
    lm_mac_t *mac;
    SETUP_TCB;

    mac = FIND_MAC(tcb->mac_q, &key->mac_addr);
    if (mac == NULL) {
        mac = add_mac(&key->mac_addr);
    }
}

lm crt_vc_cfg(key, cfg)
    lm_vc_cfg_key_t *key;
    lm_cfg_vc_t *cfg;
{
    SETUP_TCB;
}

lm crt_pv_cfg(key, cfg)
    lm_pv_cfg_key_t *key;
    lm_cfg_pv_t *cfg;
{
    lm_port_vlan_t tmp;
    lm_port_t *port;
    lm_vlan_t *vlan;
    lm_port_vlan_t *pv;

    SETUP_TCB;

    tmp.vlan_id = key->vlan_id;
    tmp.port_addr = key->port_addr;
    pv = FIND_PV(tcb->pv_q, &tmp);
    if (pv == NULL) {
```

150
lm_cfg.c
-112-

```

    port = FIND_PORT(tcb->port_q, &key->port_addr);
    if (port == NULL) {
        port = add_port(&key->port_addr);
    }
    if (vlan == NULL) {
        vlan = add_vlan(key->vlan_id, tcb->dflt_mtu_size,
            tcb->dflt_num_mcasts);
    }
    pv = add_pv(&key->port_addr, key->vlan_id, cfg->mlid);
}
lm_chg_pv_cfg(pv, &cfg->mlid);
}

lm_crt_mv_cfg(key, cfg)
    lm_mv_cfg_key_t *key;
    lm_cfg_mv_t *cfg;
{
    lm_mac_vlan_t *mv;
    lm_mac_vlan_t tmp;
    lm_mac_t *mac;
    lm_vlan_t *vlan;
    SETUP_TCB;

    tmp.vlan_id = key->vlan_id;
    tmp.mac_addr = key->mac_addr;
    mv = FIND_MV(tcb->mv_q, &tmp);
    if (mv == NULL) {
        mac = FIND_MAC(tcb->mac_q, &key->mac_addr);
        if (mac == NULL) {
            mac = add_mac(&key->mac_addr);
        }
        if (vlan == NULL) {
            vlan = add_vlan(key->vlan_id, tcb->dflt_mtu_size,
                tcb->dflt_num_mcasts);
        }
        mv = add_mv(&key->mac_addr, key->vlan_id, cfg->mlid);
    }
    lm_chg_mv_cfg(mv, &cfg->mlid);
}

```

151
lm_mgmt.c
-113-

```
/* lm_mgmt.c
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED
 */
```

*****END*****/

```
#ifdef CERNEL
```

```
#include "ipc_def.h"
#include "net_def.h"
#include <global_def.h>
#include <driver.h>
```

```
#undef lm_init
```

```
#else /* ifndef CERNEL */
```

```
#include <stdint.h>
#include <global_def.h>
#include <ITC_if.h>
#include <driver.h>
#include <RT_if.h>
#include <timer.h>
#include <RT_def.h>
#include <enet_if.h>
#include <net_def.h>
```

```
#define ERRLOG printdbg
#define printf printdbg
```

```
#endif /* ifndef CERNEL */
```

```
#include "unipdu.h"
#include "nnipdus.h"
#include "altask_gl.h"
#include "slgtask_gl.h"
#include "svctask_gl.h"
#include "svc_if.h"
#include "snmp_incl.h"
#include "AAL_if.h"
#include "q.h"
#include "bits.h"
#include "lm.h"
```

```
extern lm_port_vlan_t *lm_mgmt_find_pv();
extern lm_mac_vlan_t *lm_mgmt_find_mv();
extern lm_vlan_t *lm_mgmt_find_vlan();
extern lm_port_t *lm_mgmt_find_port();
```

152
lm_mgmt.c
-114-

```

extern lm_mac_t *lm_mgmt_find_mac();
extern lm_vc_t *lm_mgmt_find_vc();

extern lm_port_vlan_t *lm_mgmt_findnext_pv();
extern lm_mac_vlan_t *lm_mgmt_findnext_mv();
extern lm_vlan_t *lm_mgmt_findnext_vlan();
extern lm_port_t *lm_mgmt_findnext_port();
extern lm_mac_t *lm_mgmt_findnext_mac();
extern lm_vc_t *lm_mgmt_findnext_vc();

lm_srvc_mgmt_get(msg)
    struct AgentMsg *msg;
{
    int      ret;

    ret = lm_do_mgmt_get(msg, FALSE);
    return (ret);
}

lm_srvc_mgmt_getnext(msg)
    struct AgentMsg *msg;
{
    int      ret;

    ret = lm_do_mgmt_get(msg, TRUE);
    return (ret);
}

lm_do_mgmt_get(msg, getnext)
    struct AgentMsg *msg;
    int      getnext;
{
    int      ret;
    struct MB      *mb;
    struct MBH     *mb_hdr;
    struct AgentMsg *tx_msg;

    ret = RT_SUCCESS;
    tx_msg = lm_cp_mgmt_msg(msg);
    mb = &tx_msg->Body;
    mb_hdr = &mb->head;
    mb_hdr->datoff = 0;
    switch (mb_hdr->ovcode) {
    case LM_GLBL_OVN:
        ret = lm_mgmt_get_glbl(tx_msg, getnext);
        break;
    case LM_ATTR_ENT_OVN:
        ret = lm_mgmt_get_attr_ent(tx_msg, getnext);
        break;
    }
}

```

153
lm_mgmt.c
-115-

```

case LM_PV_OVN:
    ret = lm_mgmt_get_pv(tx_msg, getnext);
    break;
case LM_NODE_ENT_OVN:
    ret = lm_mgmt_get_node_ent(tx_msg, getnext);
    break;
case LM_MAC_ENT_OVN:
    ret = lm_mgmt_get_mac_ent(tx_msg, getnext);
    break;
case LM_MP_ENT_OVN:
    ret = lm_mgmt_get_mp_ent(tx_msg, getnext);
    break;
case LM_PM_ENT_OVN:
    ret = lm_mgmt_get_pm_ent(tx_msg, getnext);
    break;
case LM_VC_ENT_OVN:
    ret = lm_mgmt_get_vc_ent(tx_msg, getnext);
    break;
default:
    ret = IRT_SUCCESS;
    mb_hdr->rcode = ER_GENERIC;
    goto err_exit;
    break;
}
ret = lm_send_mgmt_rsp(tx_msg);
return (ret);

```

```

err_exit:
    lm_send_mgmt_rsp(tx_msg);
    return (ret);
}

```

```

lm_mgmt_get_glbl(msg, getnext)
struct AgentMsg *msg;
int getnext;
{
    int ret;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct OS tmp_os;
    int num_vlans;
    int num_ports;
    int num_macs;
    int num_mvs;
    int num_pvs;
    int num_vcs;
    SETUP_TCB;

    if (CHK_VB(LM_VB_MSGS)) {
        printf("getting globals\r\n");
    }
}

```

154
lm_mgmt.c
-116-

```

    }
    num_vlans = QUEUE_LEN(tcb->vlan_q);
    num_ports = QUEUE_LEN(tcb->port_q);
    num_macs = QUEUE_LEN(tcb->mac_q);
    num_mvs = QUEUE_LEN(tcb->mvs_q);
    num_pvs = QUEUE_LEN(tcb->pvs_q);
    num_vcs = QUEUE_LEN(tcb->vc_q);
    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    tmp_os.length = sizeof(tcb->nac_atm_addr);
    bcopy(&tcb->nac_atm_addr, tmp_os.buffer, tmp_os.length);
    MB_wp_OS(mb, LM_NAC_ADDR_PIX, &tmp_os);
    tmp_os.length = sizeof(tcb->bld_bits);
    bcopy(tcb->bld_bits, tmp_os.buffer, tmp_os.length);
    MB_wp_OS(mb, LM_BID_PIX, &tmp_os);
    tmp_os.length = sizeof(tcb->mlid_bits);
    bcopy(tcb->mlid_bits, tmp_os.buffer, tmp_os.length);
    MB_wp_OS(mb, LM_MLID_BITS_PIX, &tmp_os);
    MB_wp_INT(mb, LM_DFLT_MTU_PIX, &tcb->dflt_mtu_size);
    MB_wp_INT(mb, LM_DFLT_MCAST_PIX, &tcb->dflt_num_mcasts);
    MB_wp_INT(mb, LM_NUM_VLANS_PIX, &num_vlans);
    MB_wp_INT(mb, LM_NUM_MACS_PIX, &num_macs);
    MB_wp_INT(mb, LM_NUM_PORTS_PIX, &num_ports);
    MB_wp_INT(mb, LM_NUM_MVS_PIX, &num_mvs);
    MB_wp_INT(mb, LM_NUM_PVS_PIX, &num_pvs);
    MB_wp_INT(mb, LM_NUM_VCS_PIX, &num_vcs);
    return (ret);

err_exit:
    return (ret);
}

```

```

lm_mgmt_get_attr_ent(msg, getnext)
    struct AgentMsg *msg;
    int getnext;
{
    int ret;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct LM_ATTR_ENT_IDX *idx;
    lm_vlan_t *vlan;
    struct OS tmp_os;
    struct LM_ATTR_ENT_IDX tmp_idx;
    int num_macs;
    int num_ports;
    SETUP_TCB;

    ret = RT_SUCCESS;

```

155
lm_mgmt.c
-117-

```

mb = &msg->Body;
mb_hdr = &mb->head;
idx = (struct LM_ATTR_ENT_IDX *) mb_hdr->lid;
vlan = lm_mgmt_find_vlan(idx);
if (vlan == NULL) {
    if (getnext) {
        vlan = lm_mgmt_findnext_vlan(idx);
        if (vlan == NULL) {
            ret = IRT_SUCCESS;
            mb_hdr->rcode = ER_NOSUCHNAME;
            goto err_exit;
        }
    } else {
        ret = IRT_SUCCESS;
        mb_hdr->rcode = ER_NOSUCHNAME;
        goto err_exit;
    }
}
if (CHK_VB(LM_VB_MSGS)) {
    printf("getting stats on vlan %d\n", vlan->vlan_id);
}
num_macs = QUEUE_LEN(vlan->mv_q);
num_ports = QUEUE_LEN(vlan->pv_q);

lm_cvt_vlan_idx(&vlan->vlan_id, &tmp_idx);
*idx = tmp_idx;
lm_cvt_idx_os(tmp_idx.LM_VLAN, LM_NUM_ELEM(tmp_idx.LM_VLAN), &tmp_os);
MB_wp_OS(mb, LM_VLAN_PIX, &tmp_os);

tmp_os.length = strlen(vlan->vlan_name);
bcopy(vlan->vlan_name, tmp_os.buffer, tmp_os.length);
MB_wp_OS(mb, LM_VLAN_NAME_PIX, &tmp_os);

MB_wp_INT(mb, LM_NUM_MCAST_PIX, &vlan->num_mcasts);
MB_wp_INT(mb, LM_MTU_SIZE_PIX, &vlan->mtu_size);
MB_wp_INT(mb, LM_MLID_PIX, &vlan->dflt_mlid);
MB_wp_INT(mb, LM_ATTR_NUM_PORTS_PIX, &num_ports);
MB_wp_INT(mb, LM_ATTR_NUM_MACS_PIX, &num_macs);

tmp_os.length = sizeof(vlan->mid_bits);
bcopy(vlan->mid_bits, tmp_os.buffer, tmp_os.length);
MB_wp_OS(mb, LM_MID_BITS_PIX, &tmp_os);

return (ret);

err_exit:
return (ret);
}

lm_mgmt_get_pv(msg, getnext)

```

156
lm_mgmt.c
-118-

```

struct AgentMsg *msg;
int      getnext;
{
    int      ret;
    struct MB      *mb;
    struct MBH      *mb_hdr;
    struct LM_PV_IDX *idx;
    struct LM_PV_IDX tmp_idx;
    lm_port_vlan_t *pv;
    struct OS      tmp_os;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    idx = (struct LM_PV_IDX *) mb_hdr->Id;
    pv = lm_mgmt_find_pv(idx);
    if (pv == NULL) {
        if (getnext) {
            pv = lm_mgmt_findnext_pv(idx);
            if (pv == NULL) {
                ret = IRT_SUCCESS;
                mb_hdr->rcode = ER_NOSUCHNAME;
                goto err_exit;
            }
        } else {
            ret = IRT_SUCCESS;
            mb_hdr->rcode = ER_GENERIC;
            goto err_exit;
        }
    }
    if (CHK_VB(LM_VB_MSGS)) {
        printf("getting stats on pv = %r\n", pv);
        print_pv(pv, 0);
    }
    lm_cvt_pv_idx(pv, &tmp_idx);
    *idx = tmp_idx;
    MB_wp_INT(mb, LM_PV_SHELF_PIX, &tmp_idx.LM_PV_SHELF);
    MB_wp_INT(mb, LM_PV_CARD_PIX, &tmp_idx.LM_PV_CARD);
    MB_wp_INT(mb, LM_PV_PORT_PIX, &tmp_idx.LM_PV_PORT);
    MB_wp_INT(mb, LM_PV_MLID_PIX, &pv->mlid);
    lm_cvt_idx_os(tmp_idx.LM_PV_VLAN, LM_NUM_ELEM(tmp_idx.LM_PV_VLAN),
        &tmp_os);
    MB_wp_OS(mb, LM_PV_VLAN_PIX, &tmp_os);
    return (ret);
err_exit:
    return (ret);
}

```

157
lm_mgmt.c
-119-

```

lm_mgmt_get_node_ent(msg, getnext)
    struct AgentMsg *msg;
    int             getnext;
{
    int             ret;
    struct MB       *mb;
    struct MBH      *mb_hdr;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;

    print_tcb(tcb, 0);
    ret = IRT_SUCCESS;
    mb_hdr->rcode = ER_GENERIC;
    goto err_exit;

    return (ret);

err_exit:
    return (ret);
}

lm_mgmt_get_mac_ent(msg, getnext)
    struct AgentMsg *msg;
    int             getnext;
{
    int             ret;
    struct MB       *mb;
    struct MBH      *mb_hdr;
    struct LM_MAC_ENT_IDX *idx;
    struct LM_MAC_ENT_IDX tmp_idx;
    lm_mac_vlan_t   tmp_mv;
    lm_mac_addr_t   mac_addr;
    lm_mac_vlan_t   *mv;
    struct OS       tmp_os;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    idx = (struct LM_MAC_ENT_IDX *) mb_hdr->lld;
    mv = lm_mgmt_find_mv(idx);
    if (mv == NULL) {
        if (getnext) {
            mv = lm_mgmt_findnext_mv(idx);
            if (mv == NULL) {
                ret = !RT_SUCCESS;
            }
        }
    }
}

```

/58

lm_mgmt.c
-120-

```

        mb_hdr->rcode = ER_NOSUCHNAME;
        goto err_exit;
    }
    else {
        ret = IRT_SUCCESS;
        mb_hdr->rcode = ER_GENERIC;
        goto err_exit;
    }
}
if (CHK_VB(LM_VB_MSGS)) {
    printf("getting stats on mv = ");
    print_mv(mv, 0);
}
lm_cvt_mv_idx(mv, &tmp_idx);
*idx = tmp_idx;
lm_cvt_idx_os(tmp_idx.LM_MAC_VLAN, LM_NUM_ELEM(tmp_idx.LM_MAC_VLAN),
    &tmp_os);
MB_wp_OS(mb, LM_MAC_VLAN_PIX, &tmp_os);

lm_cvt_idx_os(tmp_idx.LM_MAC_ADDR, LM_NUM_ELEM(tmp_idx.LM_MAC_ADDR),
    &tmp_os);
MB_wp_OS(mb, LM_MAC_ADDR_PIX, &tmp_os);
MB_wp_INT(mb, LM_MAC_MLID_PIX, &mv->mlid);
return (ret);

err_exit:
return (ret);
}

lm_mgmt_get_mp_ent(msg, getnext)
struct AgentMsg *msg;
int getnext;
{
    int ret;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct LM_MP_ENT_IDX *idx;
    struct LM_MP_ENT_IDX tmp_idx;
    lm_mac_addr_t mac_addr;
    lm_mac_t *mac;
    lm_port_t *port;
    lm_port_addr_t port_addr;
    struct OS tmp_os;
    int shelf;
    int slot;
    int port_num;
    int num_vlans;
    SETUP_TCB;

    ret = RT_SUCCESS;

```

159

lm_mgmt.c
-121-

```

mb = &msg->Body;
mb_hdr = &mb->head;
idx = (struct LM_MP_ENT_IDX *) mb_hdr->lid;
mac = lm_mgmt_find_mac(idx);
if (mac == NULL) {
    if (getnext) {
        mac = lm_mgmt_findnext_mac(idx);
        if (mac == NULL) {
            ret = IRT_SUCCESS;
            mb_hdr->rcode = ER_NOSUCHNAME;
            goto err_exit;
        }
    } else {
        ret = IRT_SUCCESS;
        mb_hdr->rcode = ER_GENERIC;
        goto err_exit;
    }
}
if (CHK_VB(LM_VB_MSGS)) {
    printf("getting stats on mac = %s\r\n",
        sprint_mac_addr(&mac->mac_addr));
}
num_vlans = QUEUE_LEN(mac->mv_q);

lm_cvt_mac_idx(&mac->mac_addr, &tmp_idx);
*idx = tmp_idx;
lm_cvt_idx_os(tmp_idx.LM_MP_MAC, LM_NUM_ELEM(tmp_idx.LM_MP_MAC),
    &tmp_os);
MB_wp_OS(mb, LM_MP_MAC_PIX, &tmp_os);

LM_CLR_PORT_ADDR(&port_addr);
if (mac->port != NULL) {
    port = mac->port;
    LM_INIT_PORT_ADDR(&port_addr, tcb->my_node,
        port->port_addr.aa_shelf + 1,
        port->port_addr.aa_slot + 1,
        port->port_addr.aa_port + 1);
}
shelf = port_addr.aa_shelf;
slot = port_addr.aa_slot;
port_num = port_addr.aa_port;

MB_wp_INT(mb, LM_MP_SHELF_PIX, &shelf);
MB_wp_INT(mb, LM_MP_CARD_PIX, &slot);
MB_wp_INT(mb, LM_MP_PORT_PIX, &port_num);
MB_wp_INT(mb, LM_MP_NUM_VLANS_PIX, &num_vlans);
tmp_os.length = sizeof(mac->mlid_bits);
bcopy(mac->mlid_bits, tmp_os.buffer, tmp_os.length);
MB_wp_OS(mb, LM_MP_MLID_BITS_PIX, &tmp_os);

```

160
lm_mgmt.c
-122-

```

    return (ret);

err_exit:
    return (ret);
}

lm_mgmt_get_vc_ent(msg, getnext)
    struct AgentMsg *msg;
    int getnext;
{
    int ret;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct LM_VC_ENT_IDX *idx;
    struct LM_VC_ENT_IDX tmp_idx;
    lm_vc_addr_t vc_addr;
    lm_vc_t *vc;
    struct OS tmp_os;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    idx = (struct LM_VC_ENT_IDX *) mb_hdr->iid;
    vc = lm_mgmt_find_vc(idx);
    if (vc == NULL) {
        if (getnext) {
            vc = lm_mgmt_findnext_vc(idx);
            if (vc == NULL) {
                ret = IRT_SUCCESS;
                mb_hdr->rcode = ER_NOSUCHNAME;
                goto err_exit;
            }
        } else {
            ret = IRT_SUCCESS;
            mb_hdr->rcode = ER_GENERIC;
            goto err_exit;
        }
    }
    if (CHK_VB(LM_VB_MSGS)) {
        printf("getting stats on vc = %s\r\n",
            sprint_vc_addr(&vc->vc_addr));
    }
    lm_cvt_vc_idx(&vc->vc_addr, &tmp_idx);
    *idx = tmp_idx;
    lm_cvt_idx_os(tmp_idx.LM_VC_VLAN, LM_NUM_ELEM(tmp_idx.LM_VC_VLAN),
        &tmp_os);
    MB_wp_OS(mb, LM_VC_VLAN_PIX, &tmp_os);
    lm_cvt_idx_os(tmp_idx.LM_VC_MAC, LM_NUM_ELEM(tmp_idx.LM_VC_MAC),

```

161
lm_mgmt.c
-123-

```

        &tmp_os);
    MB_wp_OS(mb, LM_VC_MAC_PIX, &tmp_os);

    MB_wp_INT(mb, LM_VC_REF_CNT_PIX, &vc->ref_cnt);
    MB_wp_INT(mb, LM_VC_BID_PIX, &vc->bid);
    return (ret);

err_exit:
    return (ret);
}

lm_mgmt_get_pm_ent(msg, getnext)
    struct AgentMsg *msg;
    int getnext;
{
    int ret;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct LM_PM_ENT_IDX *idx;
    struct LM_PM_ENT_IDX tmp_idx;
    lm_mac_addr_t mac_addr;
    lm_mac_t *mac;
    lm_port_t *port;
    lm_port_addr_t port_addr;
    struct OS tmp_os;
    int shelf;
    int slot;
    int port_num;
    int num_vlans;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    idx = (struct LM_PM_ENT_IDX *) mb_hdr->iid;
    port = lm_mgmt_find_port(idx);
    if (port == NULL) {
        if (getnext) {
            port = lm_mgmt_findnext_port(idx);
            if (port == NULL) {
                ret = IRT_SUCCESS;
                mb_hdr->rcode = ER_NOSUCHNAME;
                goto err_exit;
            }
        } else {
            ret = IRT_SUCCESS;
            mb_hdr->rcode = ER_GENERIC;
            goto err_exit;
        }
    }
}

```

162
lm_mgmt.c
-124-

```

num_vlans = QUEUE_LEN(port->pv_q);

if (CHK_VB(LM_VB_MSGS)) {
    printf("getting stats on port %s\r\n",
        sprint_port_addr(&port->port_addr));
}
lm_cvt_port_idx(&port->port_addr, &tmp_idx);
*idx = tmp_idx;
port_addr = port->port_addr;
LM_CLR_MAC_ADDR(&mac_addr);
if (port->mac != NULL) {
    mac = port->mac;
    mac_addr = mac->mac_addr;
}
shelf = port_addr.aa_shelf + 1;
slot = port_addr.aa_slot + 1;
port_num = port_addr.aa_port + 1;
MB_wp_INT(mb, LM_PM_SHELF_PIX, &shelf);
MB_wp_INT(mb, LM_PM_CARD_PIX, &slot);
MB_wp_INT(mb, LM_PM_PORT_PIX, &port_num);
MB_wp_INT(mb, LM_PM_NUM_VLANS_PIX, &num_vlans);
tmp_os.length = sizeof(mac_addr) - 2;
bcopy((char *) &mac_addr + 2, tmp_os.buffer, tmp_os.length);
MB_wp_OS(mb, LM_PM_MAC_PIX, &tmp_os);
tmp_os.length = sizeof(port->mlid_bits);
bcopy(port->mlid_bits, tmp_os.buffer, tmp_os.length);
MB_wp_OS(mb, LM_PM_MLID_BITS_PIX, &tmp_os);
return (ret);

err_exit:
return (ret);
}

lm_svc_mgmt_validate(msg)
struct AgentMsg *msg;
{
    int      ret;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct AgentMsg *tx_msg;

    ret = RT_SUCCESS;
    tx_msg = lm_cp_mgmt_msg(msg);
    mb = &tx_msg->Body;
    mb_hdr = &mb->head;
    switch (mb_hdr->ovcode) {
        case LM_GLBL_OVN:
            ret = lm_mgmt_val_glbl(tx_msg);
            break;
        case LM_ATTR_ENT_OVN:
    
```

163
lm_mgmt.c
-125-

```

    ret = lm_mgmt_val_attr_ent(tx_msg);
    break;
case LM_PV_OVN:
    ret = lm_mgmt_val_pv(tx_msg);
    break;
case LM_NODE_ENT_OVN:
    ret = lm_mgmt_val_node_ent(tx_msg);
    break;
case LM_MAC_ENT_OVN:
    ret = lm_mgmt_val_mac_ent(tx_msg);
    break;
case LM_MP_ENT_OVN:
    ret = lm_mgmt_val_mp_ent(tx_msg);
    break;
case LM_PM_ENT_OVN:
    ret = lm_mgmt_val_pm_ent(tx_msg);
    break;
case LM_VC_ENT_OVN:
    ret = lm_mgmt_val_vc_ent(tx_msg);
    break;
default:
    ret = IRT_SUCCESS;
    mb_hdr->rcode = ER_GENERIC;
    goto err_exit;
    break;
}
ret = lm_send_mgmt_rsp(tx_msg);
return (ret);

err_exit:
    lm_send_mgmt_rsp(tx_msg);
    return (ret);
}

```

```

lm_mgmt_val_attr_ent(msg)
struct AgentMsg *msg;
{
    int ret;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct LM_ATTR_ENT_IDX *idx;
    lm_vlan_t *vlan;
    lm_vlan_id_t vlan_id;
    tUINT32 mtu_size;
    tUINT32 num_mcasts;
    tUINT8 got_mtu_size;
    tUINT8 got_num_mcasts;
    SETUP_TCB;

    ret = RT_SUCCESS;
}

```

164
lm_mgmt.c
-126-

```

    mb = &msg->Body;
    mb_hdr = &mb->head;
    mtu_size = num_mcasts = -1;
    idx = (struct LM_ATTR_ENT_IDX *) mb_hdr->iid;
    vian = lm_mgmt_find_vian(idx);
    if (vian == NULL) {
    } else {
        got_num_mcasts = MB_dop(mb, LM_NUM_MCAST_PIX);
        got_mtu_size = MB_dop(mb, LM_MTU_SIZE_PIX);
        num_mcasts = got_num_mcasts ?
            MB_cp_INT(mb, LM_NUM_MCAST_PIX, &num_mcasts) : -1;
        mtu_size = got_mtu_size ?
            MB_cp_INT(mb, LM_MTU_SIZE_PIX, &mtu_size) : -1;
    }
    return (ret);

err_exit:
    return (ret);
}

lm_mgmt_val_pv(msg)
    struct AgentMsg *msg;
{
    int         ret;
    struct MB   *mb;
    struct MBH   *mb_hdr;
    struct LM_PV_IDX *idx;
    lm_port_vian_t *pv;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    idx = (struct LM_PV_IDX *) mb_hdr->iid;
    pv = lm_mgmt_find_pv(idx);

    if (pv == NULL) {
    }
    return (ret);

err_exit:
    return (ret);
}

lm_mgmt_val_node_ent(msg)
    struct AgentMsg *msg;
{
    int         ret;
    struct MB   *mb;

```

165
lm_mgmt.c
-127-

```
struct MBH    *mb_hdr;
SETUP_TCB;
```

```
ret = RT_SUCCESS;
mb = &msg->Body;
mb_hdr = &mb->head;
```

```
ret = IRT_SUCCESS;
mb_hdr->rcode = ER_GENERIC;
goto err_exit;
```

```
return (ret);
```

```
err_exit:
return (ret);
}
```

```
lm_mgmt_val_mac_ent(msg)
struct AgentMsg *msg;
```

```
{
    int          ret;
    struct MB     *mb;
    struct MBH    *mb_hdr;
    lm_mac_vlan_t *mv;
    SETUP_TCB;
```

```
ret = RT_SUCCESS;
mb = &msg->Body;
mb_hdr = &mb->head;
```

```
return (ret);
```

```
err_exit:
return (ret);
}
```

```
lm_mgmt_val_glbl(msg)
struct AgentMsg *msg;
```

```
{
    int          ret;
    SETUP_TCB;
```

```
ret = RT_SUCCESS;
return (ret);
}
```

```
lm_mgmt_val_mp_ent(msg)
struct AgentMsg *msg;
```

```
{
    int          ret;
```

166
lm_mgmt.c
-128-

```

struct MB      *mb;
struct MBH     *mb_hdr;
struct LM_MP_ENT_IDX *idx;
struct LM_MP_ENT_IDX tmp_idx;
lm_mac_addr_t  mac_addr;
lm_mac_t       *mac;
lm_port_t      *port;
lm_port_addr_t port_addr;
struct OS      tmp_os;
int            shelf;
int            slot;
int            port_num;
SETUP_TCB;

ret = RT_SUCCESS;
mb = &msg->Body;
mb_hdr = &mb->head;
idx = (struct LM_MP_ENT_IDX *) mb_hdr->lld;
mac = lm_mgmt_find_mac(idx);
if (mac == NULL) {
} else {
}
return (ret);

```

```

err_exit:
return (ret);
}

```

```

lm_mgmt_val_pm_ent(msg)
struct AgentMsg *msg;
{
    int      ret;
    struct MB      *mb;
    struct MBH     *mb_hdr;
    struct LM_PM_ENT_IDX *idx;
    struct LM_PM_ENT_IDX tmp_idx;
    lm_mac_addr_t  mac_addr;
    lm_mac_t       *mac;
    lm_port_t      *port;
    lm_port_addr_t port_addr;
    struct OS      tmp_os;
    int            shelf;
    int            slot;
    int            port_num;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;

```

167
lm_mgmt.c
-129-

```

idx = (struct LM_PM_ENT_IDX *) mb_hdr->iid;
port = lm_mgmt_find_port(idx);
if (port == NULL) {
} else {
}
return (ret);

```

```

err_exit:
return (ret);
}

```

```

lm_mgmt_val_vc_ent(msg)
struct AgentMsg *msg;
{
    int          ret;
    struct MB     *mb;
    struct MBH    *mb_hdr;
    struct LM_VC_ENT_IDX *idx;
    struct LM_VC_ENT_IDX tmp_idx;
    lm_vc_addr_t  vc_addr;
    lm_vc_t       *vc;
    struct OS     tmp_os;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    idx = (struct LM_VC_ENT_IDX *) mb_hdr->iid;
    vc = lm_mgmt_find_vc(idx);
    if (vc == NULL) {
    } else {
    }
    return (ret);
}

```

```

err_exit:
return (ret);
}

```

```

lm_srvc_mgmt_commit(msg)
struct AgentMsg *msg;
{
    int          ret;
    struct MB     *mb;
    struct MBH    *mb_hdr;
    struct AgentMsg *tx_msg;

    ret = RT_SUCCESS;
    tx_msg = lm_cp_mgmt_msg(msg);
    mb = &tx_msg->Body;
    mb_hdr = &mb->head;
}

```

128
lm_mgmt.c
-130-

```

switch (mb_hdr->ovcode) {
case LM_GLBL_OVN:
    ret = lm_mgmt_cmt_glbl(tx_msg);
    break;
case LM_ATTR_ENT_OVN:
    ret = lm_mgmt_cmt_attr_ent(tx_msg);
    break;
case LM_PV_OVN:
    ret = lm_mgmt_cmt_pv(tx_msg);
    break;
case LM_NODE_ENT_OVN:
    ret = lm_mgmt_cmt_node_ent(tx_msg);
    break;
case LM_MAC_ENT_OVN:
    ret = lm_mgmt_cmt_mac_ent(tx_msg);
    break;
case LM_MP_ENT_OVN:
    ret = lm_mgmt_cmt_mp_ent(tx_msg);
    break;
case LM_PM_ENT_OVN:
    ret = lm_mgmt_cmt_pm_ent(tx_msg);
    break;
case LM_VC_ENT_OVN:
    ret = lm_mgmt_cmt_vc_ent(tx_msg);
    break;
default:
    ret = IRT_SUCCESS;
    mb_hdr->rcode = ER_GENERIC;
    goto err_exit;
    break;
}
ret = lm_send_mgmt_rsp(tx_msg);
return (ret);

err_exit:
    lm_send_mgmt_rsp(tx_msg);
    return (ret);
}

lm_mgmt_cmt_glbl(msg)
struct AgentMsg *msg;
{
    int      ret;
    int      i;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct OS tmp_os;
    tUINT32 *dflt_mtu_size;
    tUINT32 *dflt_num_mcasts;

```

169
lm_mgmt.c
-131-

```

tATMADDR      *nac_addr;
bits_t        *bid_bits;
bits_t        *mlid_bits;
tUINT32       dflt_mtu_size_a;
tUINT32       dflt_num_mcasts_a;
tATMADDR      nac_addr_a;
bits_t        bid_bits_a[SIZE_BID_BITS];
bits_t        mlid_bits_a[SIZE_MLID_BITS];
SETUP_TCB;

ret = RT_SUCCESS;
mb = &msg->Body;
mb_hdr = &mb->head;
dflt_mtu_size = dflt_num_mcasts = nac_addr =
    bid_bits = mlid_bits = NULL;
if (MB_dop(mb, LM_NAC_ADDR_PIX)) {
    MB_cp_OS(mb, LM_NAC_ADDR_PIX, &tmp_os);
    if (tmp_os.length != sizeof(nac_addr_a)) {
        ret = IRT_SUCCESS;
        mb_hdr->rcode = ER_GENERIC;
        goto err_exit;
    }
    bcopy(tmp_os.buffer, &nac_addr_a, sizeof(nac_addr_a));
    nac_addr = &nac_addr_a;
}
if (MB_dop(mb, LM_BID_PIX)) {
    bzero(bid_bits_a, sizeof(bid_bits_a));
    MB_cp_OS(mb, LM_BID_PIX, &tmp_os);
    if (tmp_os.length > sizeof(bid_bits_a)) {
        ret = IRT_SUCCESS;
        mb_hdr->rcode = ER_GENERIC;
        goto err_exit;
    }
    bcopy(tmp_os.buffer, bid_bits_a, tmp_os.length);
    bid_bits = &bid_bits_a;
}
if (MB_dop(mb, LM_DFLT_MTU_PIX)) {
    MB_cp_INT(mb, LM_DFLT_MTU_PIX, &dflt_mtu_size_a);
    dflt_mtu_size = &dflt_mtu_size_a;
}
if (MB_dop(mb, LM_DFLT_MCAST_PIX)) {
    MB_cp_INT(mb, LM_DFLT_MCAST_PIX, &dflt_num_mcasts_a);
    dflt_num_mcasts = &dflt_num_mcasts_a;
}
if (MB_dop(mb, LM_MLID_BITS_PIX)) {
    bzero(mlid_bits_a, sizeof(mlid_bits_a));
    MB_cp_OS(mb, LM_MLID_BITS_PIX, &tmp_os);
    if (tmp_os.length > sizeof(mlid_bits_a)) {
        ret = IRT_SUCCESS;
        mb_hdr->rcode = ER_GENERIC;
    }
}

```

170
lm_mgmt.c
-132-

```

        goto err_exit;
    }
    bcopy(tmp_os.buffer, mlid_bits_a, tmp_os.length);
    mlid_bits = &mlid_bits_a;
}
lm_chg_glbl_cfg(dflt_mtu_size, dflt_num_mcasts, nac_addr, bld_bits,
    mlid_bits);
return (ret);

err_exit:
return (ret);
}

lm_mgmt_cmt_attr_ent(msg)
    struct AgentMsg *msg;
{
    int          ret;
    int          tmp_ret;
    int          a_r;
    struct MB    *mb;
    struct MBH   *mb_hdr;
    struct LM_ATTR_ENT_IDX *idx;
    qlink_t      *link;
    lm_mac_t     *mac;
    lm_mac_vlan_t *mv;
    lm_vlan_t     *vlan;
    lm_vlan_id_t  vlan_id;
    tUINT32      *mtu_size;
    tUINT32      *num_mcasts;
    tUINT32      *dflt_mlid;
    char         *vlan_name;
    tUINT32      mtu_size_a;
    tUINT32      num_mcasts_a;
    tUINT32      dflt_mlid_a;
    char         vlan_name_a[LM_MAX_VLAN_NAME];
    int          vlan_name_len;
    struct OS    tmp_os;
    int          chngd;
    SETUP_TCB;

    ret = RT_SUCCESS;
    chngd = FALSE;
    mb = &msg->Body;
    mb_hdr = &mb->head;

    a_r = MB_cmp_bx(mb, LM_VLAN_PIX, LM_VLAN_IXO, LM_VLAN_IXL);

    dflt_mlid = num_mcasts = mtu_size = vlan_name = NULL;

    num_mcasts_a = tcb->dflt_num_mcasts;

```

171
lm_mgmt.c
-133-

```

mtu_size_a = tcb->dflt_mtu_size;
dflt_mlid_a = LM_MAX_MLID;
tmp_os.length = 3;
bcopy("bob", tmp_os.buffer, tmp_os.length);
if (MB_dop(mb, LM_MLID_PIX)) {
    MB_cp_INT(mb, LM_MLID_PIX, &dflt_mlid_a);
    dflt_mlid = &dflt_mlid_a;
}
if (MB_dop(mb, LM_NUM_MCAST_PIX)) {
    MB_cp_INT(mb, LM_NUM_MCAST_PIX, &num_mcasts_a);
    num_mcasts = &num_mcasts_a;
}
if (MB_dop(mb, LM_MTU_SIZE_PIX)) {
    MB_cp_INT(mb, LM_MTU_SIZE_PIX, &mtu_size_a);
    mtu_size = &mtu_size_a;
}
if (MB_dop(mb, LM_VLAN_NAME_PIX)) {
    MB_cp_OS(mb, LM_VLAN_NAME_PIX, &tmp_os);
    if (tmp_os.length > sizeof(vlan_name_a) - 1) {
        tmp_os.length = sizeof(vlan_name_a) - 1;
    }
    vlan_name = &vlan_name_a;
}
bcopy(tmp_os.buffer, vlan_name_a, tmp_os.length);
vlan_name_a[tmp_os.length] = 0;

idx = (struct LM_ATTR_ENT_IDX *) mb_hdr->lid;
vlan = lm_mgmt_find_vlan(idx);
if (vlan == NULL && a_r == MB_IXP_CREATE) {
    lm_cvt_idx_vlan(idx->LM_VLAN, &vlan_id);
    vlan = add_vlan(vlan_id, mtu_size_a, num_mcasts_a,
        vlan_name_a);
} else if (vlan != NULL && a_r == MB_IXP_DELETE) {
    vlan_id = vlan->vlan_id;
    free_vlan(vlan);
    vlan = NULL;
}
if (vlan != NULL) {
    lm_chg_vlan_cfg(vlan, dflt_mlid, num_mcasts, mtu_size,
        vlan_name);
}
if (vlan != NULL) {
    print_vlan(vlan, 0);
} else {
    printf("tried to add vlan %d, a_r = %d\r\n", vlan_id, a_r);
    return (ret);
}

err_exit:
    return (ret);
}

```

172
lm_mgmt.c
-134-

```

lm_mgmt_cmt_pv(msg)
struct AgentMsg *msg;
{
    int      ret;
    int      a_r;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct LM_PV_IDX *idx;
    lm_port_t *port;
    lm_vlan_t *vlan;
    lm_port_addr_t port_addr;
    lm_vlan_id_t vlan_id;
    tUINT32 *mlid;
    tUINT32 mlid_a;
    lm_port_vlan_t *pv;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    mlid = NULL;
    a_r = MB_cmp_b(mb, LM_PV_VLAN_PIX, LM_PV_VLAN_IXO, LM_PV_VLAN_IXL);
    idx = (struct LM_PV_IDX *) mb_hdr->lid;
    pv = lm_mgmt_find_pv(idx);
    if (pv == NULL && a_r == MB_IXP_CREATE) {
        lm_cvt_idx_port(&idx->LM_PV_SHELF, &port_addr);
        lm_cvt_idx_vlan(&idx->LM_PV_VLAN, &vlan_id);
        port = FIND_PORT(tcb->port_q, &port_addr);
        vlan = FIND_VLAN(tcb->vlan_q, vlan_id);
        if (port == NULL) {
            port = add_port(&port_addr);
        }
        if (vlan == NULL) {
            vlan = add_vlan(vlan_id, tcb->dflt_mtu_size,
                           tcb->dflt_num_mcasts);
        }
        pv = add_pv(&port_addr, vlan_id, vlan->dflt_mlid);
        if (pv == NULL) {
            ret = IRT_SUCCESS;
            mb_hdr->rcode = ER_GENERIC;
            goto err_exit;
        }
    }
    else if (pv != NULL && a_r == MB_IXP_DELETE) {
        free_pv(pv);
        pv = NULL;
    }
    if (pv != NULL) {
        if (MB_dop(mb, LM_PV_MLID_PIX) {
            MB_cp_INT(mb, LM_PV_MLID_PIX, &mlid_a);
        }
    }
}

```

172
lm_mgmt.c
-135-

```

        mld = &mld_a;
    }
} else {
    ret = IRT_SUCCESS;
    mb_hdr->rcode = ER_GENERIC;
    goto err_exit;
}
if (pv != NULL) {
    lm_chg_pv_cfg(pv, mld);
}
return (ret);

err_exit:
return (ret);
}

lm_mgmt_cmt_node_ent(msg)
struct AgentMsg *msg;
{
    int      ret;
    struct MB  *mb;
    struct MBH *mb_hdr;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;

    ret = IRT_SUCCESS;
    mb_hdr->rcode = ER_GENERIC;
    goto err_exit;

    return (ret);

err_exit:
return (ret);
}

lm_mgmt_cmt_mac_ent(msg)
struct AgentMsg *msg;
{
    int      ret;
    int      a_r; /* add/remove flag */
    struct LM_MAC_ENT_IDX tmp_idx;
    struct MB  *mb;
    struct MBH *mb_hdr;
    struct LM_MAC_ENT_IDX *idx;
    lm_mac_addr_t mac_addr;
    lm_vlan_id_t  vlan_id;
    tUINT32      *mld;

```

174
lm_mgmt.c
-136-

```

tUINT32      mld a;
lm_mac_vlan_t *mv;
lm_mac_t     *mac;
lm_port_t    *port;
lm_vlan_t    *vlan;
SETUP_TCB;

ret = RT_SUCCESS;
mb = &msg->Body;
mb_hdr = &mb->head;
mac = NULL;
mld = NULL;
idx = (struct LM_MAC_ENT_IDX *) mb_hdr->lid;
a_r = MB_cmp_idx(mb, LM_MAC_VLAN_PIX, LM_MAC_VLAN_IPO, LM_MAC_VLAN_IPL);

mv = lm_mgmt_find_mv(idx);
if (mv == NULL && a_r == MB_IXP_CREATE) {
    lm_cvt_idx_mac(&idx->LM_MAC_ADDR, &mac_addr);
    lm_cvt_idx_vlan(&idx->LM_MAC_VLAN, &vlan_id);
    mac = FIND_MAC(tcb->mac_q, &mac_addr);
    vlan = FIND_VLAN(tcb->vlan_q, vlan_id);
    if (mac == NULL) {
        mac = add_mac(&mac_addr);
    }
    if (vlan == NULL) {
        vlan = add_vlan(vlan_id, tcb->dflt_mtu_size,
            tcb->dflt_num_mcasts);
    }
    mv = add_mv(&mac_addr, vlan_id, LM_MAX_MID, vlan->dflt_mld);
} else if (mv != NULL && a_r == MB_IXP_DELETE) {
    mac = mv->mac;
    free_mv(mv);
    mv = NULL;
}
if (mv != NULL) {
    mac = mv->mac;
    if (MB_dop(mb, LM_MAC_MLID_PIX)) {
        if (mac == NULL) {
            ret = IRT_SUCCESS;
            mb_hdr->rcode = ER_GENERIC;
            goto err_exit;
        }
        MB_cp_INT(mb, LM_MAC_MLID_PIX, &mld_a);
        mld = &mld_a;
    }
} else {
    ret = IRT_SUCCESS;
    mb_hdr->rcode = ER_GENERIC;
    goto err_exit;
}

```

175
lm_mgmt.c
-137-

```

    if (mv != NULL) {
        lm_chg_mv_cfg(mv, mlid);
    }
    return (ret);

err_exit:
    return (ret);
}

lm_mgmt_cmt_vc_ent(msg)
    struct AgentMsg *msg;
{
    int          ret;
    struct MB     *mb;
    struct MBH    *mb_hdr;
    struct LM_VC_ENT_IDX *idx;
    struct LM_VC_ENT_IDX tmp_idx;
    lm_vc_addr_t  vc_addr;
    lm_vc_t       *vc;
    lm_bid_t      *bid;
    tUINT32       *ref_cnt;
    lm_bid_t      bid_a;
    tUINT32       ref_cnt_a;
    struct OS     tmp_os;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    bid = ref_cnt = NULL;
    idx = (struct LM_VC_ENT_IDX *) mb_hdr->iid;
    vc = lm_mgmt_find_vc(idx);
    if (vc == NULL) {
        ret = IRT_SUCCESS;
        mb_hdr->rcode = ER_GENERIC;
        goto err_exit;
    }
    lm_cvt_vc_idx(&vc->vc_addr, &tmp_idx);
    *idx = tmp_idx;
    lm_cvt_idx_os(tmp_idx.LM_VC_VLAN, LM_NUM_ELEM(tmp_idx.LM_VC_VLAN),
        &tmp_os);
    if (MB_dop(mb, LM_VC_REF_CNT_PIX)) {
        MB_cp_INT(mb, LM_VC_REF_CNT_PIX, &ref_cnt_a);
        ref_cnt = &ref_cnt_a;
    }
    if (MB_dop(mb, LM_VC_BID_PIX)) {
        MB_cp_INT(mb, LM_VC_BID_PIX, &bid_a);
        bid = &bid_a;
    }
    lm_chg_vc_cfg(vc, bid, ref_cnt);
}

```

175
lm_mgmt.c
-138-

```

    return (ret);

err_exit:
    return (ret);
}

lm_mgmt_cmt_pm_ent(msg)
    struct AgentMsg *msg;
{
    int      ret;
    struct MB *mb;
    struct MBH *mb_hdr;
    struct LM_PM_ENT_IDX *idx;
    struct LM_PM_ENT_IDX tmp_idx;
    lm_mac_addr_t mac_addr_a;
    lm_mac_addr_t *mac_addr;
    bits_t *mlid_bits;
    bits_t mlid_bits_a[SIZE_MLID_BITS];
    lm_port_t *port;
    lm_port_addr_t port_addr;
    struct OS tmp_os;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    mac_addr = mlid_bits = NULL;
    idx = (struct LM_PM_ENT_IDX *) mb_hdr->lid;
    port = lm_mgmt_find_port(idx);
    if (port == NULL) {
        ret = IRT_SUCCESS;
        mb_hdr->rcode = ER_GENERIC;
        goto err_exit;
    }
    if (MB_dop(mb, LM_PM_MAC_PIX) {
        LM_CLR_MAC_ADDR(&mac_addr_a);
        MB_cp_OS(mb, LM_PM_MAC_PIX, &tmp_os);
        bcopy(tmp_os.buffer, (char *) &mac_addr_a + 2, tmp_os.length);
        mac_addr = &mac_addr_a;
    }
    if (MB_dop(mb, LM_PM_MLID_BITS_PIX) {
        MB_cp_OS(mb, LM_PM_MLID_BITS_PIX, &tmp_os);
        bcopy(tmp_os.buffer, mlid_bits_a, tmp_os.length);
        mlid_bits = &mlid_bits_a;
    }
    if (port != NULL) {
        lm_chg_port_cfg(port, mac_addr, mlid_bits);
    }
    return (ret);
}

```

177
lm_mgmt.c
-139-

```
err_exit:
    return (ret);
}
```

```
lm_mgmt_cmt mp_ent(msg)
    struct AgentMsg *msg;
```

```
{
    int          ret;
    int          a_r; /* add/remove flag */
    struct MB     *mb;
    struct MBH    *mb_hdr;
    struct LM_MP_ENT_IDX *idx;
    struct LM_MP_ENT_IDX tmp_idx;
    lm_mac_addr_t mac_addr;
    lm_mac_t      *mac;
    lm_port_t     *port;
    bits_t        *mlid_bits;
    bits_t        mlid_bits_a[SIZE_MLID_BITS];
    lm_port_addr_t port_addr;
    struct OS     tmp_os;
    int           shelf;
    int           slot;
    int           port_num;
    SETUP_TCB;

    ret = RT_SUCCESS;
    mb = &msg->Body;
    mb_hdr = &mb->head;
    mlid_bits = NULL;
    idx = (struct LM_MP_ENT_IDX *) mb_hdr->lid;
    a_r = MB_cmp_bx(mb, LM_MP_MAC_PIX, LM_MP_MAC_IXO, LM_MP_MAC_IXL);
    mac = lm_mgmt_find_mac(idx);
    if (mac == NULL && a_r == MB_IXP_CREATE) {
        lm_cvt_idx_mac(&idx->LM_MP_MAC, &mac_addr);
        mac = add_mac(&mac_addr);
        if (mac == NULL) {
            ret = IRT_SUCCESS;
            mb_hdr->rcode = ER_GENERIC;
            goto err_exit;
        }
    }
    if (mac != NULL && a_r == MB_IXP_DELETE) {
        free_mac(mac);
        mac = NULL;
    }
    if (mac != NULL) {
        if (MB_dop(mb, LM_MP_MLID_BITS_PIX)) {
            MB_cp_OS(mb, LM_MP_MLID_BITS_PIX, &tmp_os);
            bcopy(tmp_os.buffer, mlid_bits_a, tmp_os.length);
            mlid_bits = &mlid_bits_a;
        }
    }
}
```

178

lm_mgmt.c
-140-

```

    }
  }
  if (mac != NULL) {
    lm_chg_mac_cfg(mac, mld_bits);
  }
  return (ret);

err_exit:
  return (ret);
}

struct AgentMsg *
lm_cp_mgmt_msg(msg)
  struct AgentMsg *msg;
{
  struct AgentMsg *ret;
  int msg_len;

  msg_len = msg->Hdr.Length + ITSZ;
  ret = (struct AgentMsg *) ReqMsgMemZero(msg_len);
  if (ret == NULL)
    goto err_exit;

  bcopy(msg, ret, msg_len);
  return (ret);

err_exit:
  Crash(993, 0, 0);
}

lm_vc_t *
lm_mgmt_find_vc(idx)
  struct LM_PV_IDX *idx;
{
  lm_vc_addr_t tmp_vc;
  lm_vlan_t *vlan;
  lm_vc_t *ret;
  SETUP_TCB;

  ret = NULL;
  lm_cvt_idx_vc(idx, &tmp_vc);
  vlan = FIND_VLAN(tcb->vlan_q, tmp_vc.vlan_id);
  if (vlan != NULL) {
    ret = FIND_VC(vlan->vc_q, &tmp_vc);
  }
  return (ret);
}

lm_port_vlan_t *

```

179
lm_mgmt.c
-141-

```

lm_mgmt find_pv(idx)
{
    struct LM_PV_IDX *idx;
    {
        lm_port_vlan_t tmp_pv;
        lm_port_vlan_t *ret;
        SETUP_TCB;

        lm_cvt_idx_pv(idx, &tmp_pv);
        ret = FIND_PV(tcb->pv_q, &tmp_pv);
        return (ret);
    }
}

lm_mac_vlan_t *
lm_mgmt find_mv(idx)
{
    struct LM_MAC_ENT_IDX *idx;
    {
        lm_mac_vlan_t *ret;
        lm_mac_vlan_t tmp_mv;
        lm_mac_vlan_t *mv;
        SETUP_TCB;

        lm_cvt_idx_mv(idx, &tmp_mv);
        ret = FIND_MV(tcb->mv_q, &tmp_mv);
        return (ret);
    }
}

lm_mac_t *
lm_mgmt find_mac(idx)
{
    struct LM_MP_ENT_IDX *idx;
    {
        lm_mac_t *ret;
        lm_mac_addr_t mac_addr;
        SETUP_TCB;

        lm_cvt_idx_mac(idx->LM_MP_MAC, &mac_addr);
        ret = FIND_MAC(tcb->mac_q, &mac_addr);
        return (ret);
    }
}

lm_port_t *
lm_mgmt find_port(idx)
{
    struct LM_PM_ENT_IDX *idx;
    {
        lm_port_t *ret;
        lm_port_addr_t port_addr;
        SETUP_TCB;

        lm_cvt_idx_port(&idx->LM_PM_SHELF, &port_addr);
        ret = FIND_PORT(tcb->port_q, &port_addr);
        return (ret);
    }
}

```

180
lm_mgmt.c
-142-

```

}

lm_vlan_t *
lm_mgmt find_vlan(idx)
    struct LM_ATTR_ENT_IDX *idx;
{
    lm_vlan_t *ret;
    lm_vlan_id_t vlan_id;
    int i;
    SETUP_TCB;

    lm_cvt_idx_vlan(idx->LM_VLAN, &vlan_id);
    ret = FIND_VLAN(tcb->vlan_q, vlan_id);
    return (ret);
}

lm_vc_t *
lm_mgmt findnext_vc(idx)
    struct LM_VC_ENT_IDX *idx;
{
    lm_vc_addr_t tmp_vc;
    lm_vc_t *ret;
    lm_vlan_t *vlan;
    SETUP_TCB;

    ret = NULL;
    lm_cvt_idx_vc(idx, &tmp_vc);
    vlan = FIND_VLAN(tcb->vlan_q, tmp_vc.vlan_id);
    if (vlan != NULL) {
        ret = FINDNEXT_VC(vlan->vc_q, &tmp_vc);
    }
    if (ret == NULL) {
        vlan = FINDNEXT_VLAN(tcb->vlan_q, tmp_vc.vlan_id);
        if (vlan != NULL) {
            ret = FINDNEXT_VC(vlan->vc_q, &tmp_vc);
        }
    }
    return (ret);
}

lm_port_vlan_t *
lm_mgmt findnext_pv(idx)
    struct LM_PV_IDX *idx;
{
    lm_port_vlan_t tmp_pv;
    lm_port_vlan_t *ret;
    SETUP_TCB;

    lm_cvt_idx_pv(idx, &tmp_pv);
    ret = FINDNEXT_PV(tcb->pv_q, &tmp_pv);
}

```

181
lm_mgmt.c
-143-

```

    return (ret);
}

lm_mac_vlan_t *
lm_mgmt findnext_mv(idx)
    struct LM_MAC_ENT_IDX *idx;
{
    lm_mac_vlan_t *ret;
    lm_mac_vlan_t tmp_mv;
    lm_mac_addr_t mac_addr;
    lm_mac_vlan_t *mv;
    SETUP_TCB;

    lm_cvt_idx_mv(idx, &tmp_mv);
    ret = FINDNEXT_MV(tcb->mv_q, &tmp_mv);
    return (ret);
}

lm_mac_t *
lm_mgmt findnext_mac(idx)
    struct LM_MP_ENT_IDX *idx;
{
    lm_mac_t *ret;
    lm_mac_addr_t mac_addr;
    SETUP_TCB;

    lm_cvt_idx_mac(idx->LM_MP_MAC, &mac_addr);
    ret = FINDNEXT_MAC(tcb->mac_q, &mac_addr);
    return (ret);
}

lm_port_t *
lm_mgmt findnext_port(idx)
    struct LM_PM_ENT_IDX *idx;
{
    lm_port_t *ret;
    lm_port_addr_t port_addr;
    SETUP_TCB;

    lm_cvt_idx_port(&idx->LM_PM_SHELF, &port_addr);
    ret = FINDNEXT_PORT(tcb->port_q, &port_addr);
    return (ret);
}

lm_vlan_t *
lm_mgmt findnext_vlan(idx)
    struct LM_ATTR_ENT_IDX *idx;
{
    lm_vlan_t *ret;
    lm_vlan_id_t vlan_id;

```

182
lm_mgmt.c
-144-

```

int      i;
SETUP_TCB;

lm_cvt_idx_vlan(idx->LM_VLAN, &vlan_id);
ret = FINDNEXT_VLAN(tcb->vlan_q, vlan_id);
return (ret);
}

lm_cvt_mv_idx(mv, idx)
    lm_mac_vlan_t *mv;
    struct LM_MAC_ENT_IDX *idx;
{
    bzero(idx, sizeof(*idx));
    lm_cvt_mac_idx(&mv->mac_addr, idx->LM_MAC_ADDR);
    lm_cvt_vlan_idx(&mv->vlan_id, idx->LM_MAC_VLAN);
}

lm_cvt_pv_idx(pv, idx)
    lm_port_vlan_t *pv;
    struct LM_PV_IDX *idx;
{
    bzero(idx, sizeof(*idx));
    idx->LM_PV_SHELF = pv->port_addr.aa_shelf + 1;
    idx->LM_PV_CARD = pv->port_addr.aa_slot + 1;
    idx->LM_PV_PORT = pv->port_addr.aa_port + 1;
    lm_cvt_vlan_idx(&pv->vlan_id, idx->LM_PV_VLAN);
}

lm_cvt_port_idx(port_addr, idx)
    lm_port_addr_t *port_addr;
    struct LM_PM_ENT_IDX *idx;
{
    bzero(idx, sizeof(*idx));
    idx->LM_PM_SHELF = port_addr->aa_shelf + 1;
    idx->LM_PM_CARD = port_addr->aa_slot + 1;
    idx->LM_PM_PORT = port_addr->aa_port + 1;
}

lm_cvt_vc_idx(vc_addr, idx)
    lm_vc_addr_t *vc_addr;
    struct LM_VC_ENT_IDX *idx;
{
    bzero(idx, sizeof(*idx));
    lm_cvt_mac_idx(&vc_addr->mac_addr, idx->LM_VC_MAC);
    lm_cvt_vlan_idx(&vc_addr->vlan_id, idx->LM_VC_VLAN);
}

lm_cvt_mac_idx(mac, idx)
    lm_mac_addr_t *mac;

```

/ 23
lm_mgmt.c
-145-

```

struct LM_MP_ENT_IDX *idx;
{
    int i;

    bzero(idx, sizeof(*idx));
    for (i = ATM_FIRST_MAC; i < sizeof(lm_mac_addr_t); i++) {
        idx->LM_MP_MAC[i - ATM_FIRST_MAC] = mac->aa_byte[i];
    }
}

lm_cvt_vlan_idx(vlan_id, idx)
    lm_vlan_id_t *vlan_id;
    struct LM_ATTR_ENT_IDX *idx;
{
    bzero(idx, sizeof(*idx));
    idx->LM_VLAN[LM_VLAN_IXL - 1] = *vlan_id;
}

/**/

lm_cvt_idx_mv(idx, mv)
    struct LM_MAC_ENT_IDX *idx;
    lm_mac_vlan_t *mv;
{
    lm_cvt_idx_mac(idx->LM_MAC_ADDR, &mv->mac_addr);
    lm_cvt_idx_vlan(idx->LM_MAC_VLAN, &mv->vlan_id);
}

lm_cvt_idx_pv(idx, pv)
    struct LM_PV_IDX *idx;
    lm_port_vlan_t *pv;
{
    lm_cvt_idx_port(&idx->LM_PV_SHELF, &pv->port_addr);
    lm_cvt_idx_vlan(idx->LM_PV_VLAN, &pv->vlan_id);
}

lm_cvt_idx_vc(idx, vc_addr)
    struct LM_VC_ENT_IDX *idx;
    lm_vc_addr_t *vc_addr;
{
    lm_cvt_idx_mac(idx->LM_VC_MAC, &vc_addr->mac_addr);
    lm_cvt_idx_vlan(idx->LM_VC_VLAN, &vc_addr->vlan_id);
}

lm_cvt_idx_port(idx, port)
    struct LM_PM_ENT_IDX *idx;
    lm_port_addr_t *port;
{
    SETUP_TCB;
}

```

182/
lm_mgmt.c
-146-

```
LM_INIT_PORT_ADDR(port, tcb->my_node, idx->LM_PM_SHELF - 1,
    idx->LM_PM_CARD - 1, idx->LM_PM_PORT - 1);
}
```

```
lm_cvt_idx_mac(idx, mac)
    struct LM_MP_ENT_IDX *idx;
    lm_mac_addr_t *mac;
{
    int i;

    LM_CLR_MAC_ADDR(mac);
    for (i = ATM_FIRST_MAC; i < sizeof(lm_mac_addr_t); i++) {
        mac->aa_byte[i] = idx->LM_MP_MAC[i - ATM_FIRST_MAC];
    }
}
```

```
lm_cvt_idx_vlan(idx, vlan_id)
    struct LM_ATTR_ENT_IDX *idx;
    lm_vlan_id_t *vlan_id;
{
    *vlan_id = idx->LM_VLAN[LM_VLAN_IXL - 1];
}
```

```
lm_cvt_idx_os(idx, len, os)
    u_short *idx;
    int len;
    struct OS *os;
{
    int i;

    os->length = len;
    for (i = 0; i < len; i++) {
        os->buffer[i] = idx[i];
    }
}
```

185
lm_util.c
-147-

```

/* lm_util.c
 * COPYRIGHT 1992 ADAPTIVE CORPORATION
 * ALL RIGHTS RESERVED

 * Description:
 *      <Description of the general category of file contents>
 * Routines:
 *      <An OPTIONAL list summarizing the routines in this file>
 *****END*****/

#ifdef CERNEL

#include "ipc_def.h"
#include "net_def.h"
#include <global_def.h>
#include <driver.h>

#undef lm_init

#else /* ifndef CERNEL */

#include <stdint.h>
#include <ITC_if.h>
#include <RT_if.h>
#include <global_def.h>
#include <driver.h>
#include <timer.h>
#include <RT_def.h>
#include <enet_if.h>
#include <net_def.h>
#include <NAC_shared_def.h>

#define ERRLOG printdbg
#define printf printdbg

#endif /* ifndef CERNEL */

#include "unipdu.h"
#include "nnlpdus.h"
#include "altask_gl.h"
#include "sigtask_gl.h"
#include "svctask_gl.h"
#include "svc_if.h"
#include "snmp_incl.h"
#include "AAL_if.h"
#include "wdb_if.h"
#include "q.h"
#include "bits.h"
#include "lm.h"

```

186
lm_util.c
-148-

```
static char hex_dlg[] = "0123456789abcdef";

lm_es_cfg_resp_t *
lm_build_es_cfg_resp(mac, enq, resp_len)
    lm_mac_t *mac;
    tCFGELEM *enq;
    int *resp_len;
{
    lm_es_cfg_resp_t *ret;
    int ret_len;
    int num_paddrs;
    tPORT_CFGELEM *paddr;
    qlink_t *link;
    lm_mac_vlan_t *mv;
    lm_port_t *port;
    struct atm_addr port_addr;
    int i;
    int mlid;
    SETUP_TCB;

    ret = NULL;
    ret_len = 0;

    if (CHK_VB(LM_VB_MSGS)) {
        printf("Sending es_cfg_resp msg to mac %s\r\n",
            sprint_mac_addr(&mac->mac_addr));
    }
    port = mac->port;
    num_paddrs = -1;
    if (port != NULL) {
        for (link = HEAD_Q(mac->mv_q); link != NULL; link = link->next) {
            mv = (lm_mac_vlan_t *) link->data;
            mlid = mv->mlid;
            printf("num_paddrs = %d, mlid = %d\r\n", num_paddrs, mlid);
            num_paddrs = (num_paddrs <= mlid) ? mlid : num_paddrs;
        }
    }
    num_paddrs++;
    ret_len = SIZE_LM_ES_CFG_RESP +
        (num_paddrs - 1) * sizeof(tPORT_CFGELEM);

    ret = (lm_es_cfg_resp_t *)
        ReqMsgMemZero(ret_len);
    if (ret == NULL)
        goto err_ext;

    bzero(ret, ret_len);
    BUILD_UNI_HDRm(&ret->lm_hdr, NNI_PROTOCOL, NN_PDU_STATUS_RESP,
        LMI_STATUS_CONFIG, LMI_GLOBAL_CREF_TYPE,
        LMI_GLOBAL_CREF_VALUE);
}
```

187
lm_util.c
149

```

ret->enq = *enq;
port_addr = port->port_addr;
paddr = ret->paddr;

for (i = 0; i < num_paddrs; i++) {
    paddr[i].af_type = LMI_PORT_ADDR;
    paddr[i].af_port = port_addr;
    paddr[i].af_port_aa_lanum = i;
}

link = HEAD_Q(mac->mv_q);
for (link = HEAD_Q(mac->mv_q); link != NULL; link = link->next) {
    mv = (lm_mac_vlan_t *) link->data;
    mlid = mv->mlid;
    paddr[mlid].af_type = LMI_PORT_ADDR;
    paddr[mlid].af_mid = mv->mid;
    if (mv->vlan == NULL) {
        paddr[mlid].af_mcasts = 0;
        paddr[mlid].af_mtu = 0;
    } else {
        paddr[mlid].af_mcasts = mv->vlan->num_mcasts;
        paddr[mlid].af_mtu = mv->vlan->mtu_size;
    }
    paddr[mlid].af_port = port_addr;
    paddr[mlid].af_port_aa_lanum = mv->mlid;
}
*resp_len = ret_len;
return (ret);

err_exit:
*resp_len = 0;
return (ret);
}

lm_send_mgmt_rsp(msg)
struct AgentMsg *msg;
{
    int ret;
    SETUP_TCB;

    if (CHK_VB(LM_VB_MSGS)) {
        printf("sending a mgmt rsp\n");
    }
    ret = SendProxyMsg(msg, msg->Body.head.mbsize,
        SNMPA_MGMT_GETRESP);
    return (ret);
}

lm_send_svc_rel_req(lmi_hdr, cause)
tLMIHDR *lmi_hdr;

```

188
lm_util.c
-150-

```

    tUINT32      cause;
{
    int          ret;
    struct svcif *msg;
    tUINT32      msg_len;
    tREL_REQ     *rel_req;
    tUINT8       *rel_cause;
    tLMIHDR      *tx_lmi_hdr;
    tITC_HEADER  *itc;
    SETUP_TCB;

    if (CHK_VB(LM_VB_MSGS)) {
        printf("sending a svc_rel_req msg, cause is %d\r\n",
            cause);
    }
    ret = RT_SUCCESS;
    msg_len = SVCIF_PDU_OFFSET + sizeof(*rel_req);
    msg = (struct svcif *) ReqMsgMemZero(msg_len);
    if (msg == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    rel_req = (tREL_REQ *) &msg->lmi_hdr;
    tx_lmi_hdr = &rel_req->lmi_hdr;
    *tx_lmi_hdr = *lmi_hdr;
    tx_lmi_hdr->lh_pdu_type = SDU_RELEASE_REQ;
    rel_cause = (tUINT8 *) &rel_req->lmi_cause;
    LMI_ADD_ELEMENT(rel_cause, LMI_RELEASE_CAUSE, cause);
    ret = lm_send_svc_msg(msg, msg_len);
    return (ret);

err_exit:
    return (ret);
}

lm_send_svc_msg(itc, len)
    tITC_HEADER  *itc;
    tUINT32      len;
{
    int          ret;
    SETUP_TCB;

    if (CHK_VB(LM_VB_MSGS)) {
        printf("sending a svc msg\r\n");
    }
    BUILD_ITCH((*itc), len - IASZ, TID_SVC, 0, EX_REQUEST,
        TA_AAL_IND_RECEIVE, tcb->mytid);
    ret = SendMsg(itc);
    return (ret);
}

```

189
lm_util.c
-151-

```

lm_send_alan_cfg(prefix, atm_hdr, slot_num, active_ports, num_paddrs, paddrs)
{
    lm_prefix_t    prefix;
    lm_atm_hdr_t   atm_hdr;
    tUINT8         slot_num;
    tUINT32        active_ports;
    tUINT32        num_paddrs;
    tATMADDR       *paddrs;

    lm_alan_cfg_resp_t *resp;
    int             ret;
    int             resp_len;
    int             i;
    SETUP_TCB;

    if (CHK_VB(LM_VB_MSGS)) {
        printf("send an alan_cfg msg, prefix = 0x%x, atm_hdr = 0x%x\r\n", prefix, atm_hdr);
        printf("\tslot_num = %d, act_ports = %d, num_paddrs = %d\r\n",
            slot_num, active_ports, num_paddrs);
    }
    resp_len = SIZE_LM_ALAN_CFG_RESP + (num_paddrs - 1) * sizeof(tATMADDR);

    resp = (lm_alan_cfg_resp_t *) ReqMsgMemZero(resp_len, 0);
    if (resp == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    /* Fill in the NNSTATUS_RESP fields */
    BUILD_UNI_HDRm(&resp->lm_hdr, NNI_PROTOCOL, NN_PDU_STATUS_RESP,
        LMI_STATUS_CONFIG, LMI_GLOBAL_CREF_TYPE,
        LMI_GLOBAL_CREF_VALUE);

    /* Fill in the ALANCFG_ENQ fields */

    resp->enq.elem_type = ALAN_CFG_ENQ;
    resp->enq.slotid = slot_num;

    /* Fill in the ALANCFG_RESP fields */

    resp->resp.elem_type = ALAN_CFG_RESP;
    resp->resp.active_ports = active_ports;
    resp->resp.nac_id = tcb->nac_id;
    ATM_ADDR_COPY(resp->resp.nac_addr, tcb->nac_atm_addr);
    resp->resp.num_paddr = num_paddrs;
    for (i = 0; i < num_paddrs; i++)
        ATM_ADDR_COPY(resp->resp.paddrs[i], paddrs[i]);

    ret = AAL_DataSendNR(&tcb->my_aal_key, resp, resp_len,
        *((tUINT32 *) &prefix), *((tUINT32 *) &atm_hdr));
    return (ret);
}

```

190
lm_util.c
-152-

```

err_exit:
    if (resp != NULL)
        FreeMem(resp);
    return (ret);
}

lm_send_es_cfg_ind(mac)
    lm_mac_t *mac;
{
    int ret;
    lm_port_t *port;
    lm_prefix_t prefix;
    lm_atm_hdr_t atm_hdr;
    tCFGELEM enq;
    lm_es_cfg_resp_t *resp;
    int resp_len;
    tUINT32 tx_shelf;
    tUINT32 tx_slot;
    tUINT32 tx_port;
    tUINT32 tx_vci;
    SETUP_TCB;

    if (CHK_VB(LM_VB_MSGS)) {
        printf("sending es_cfg_ind to mac %s\r\n",
            sprint_mac_addr(&mac->mac_addr));
    }
    ret = RT_SUCCESS;

    port = mac->port;
    if (port == NULL) {
        ret = IRT_SUCCESS;
        goto err_exit;
    }
    bzero(&enq, sizeof(enq));
    tx_shelf = port->port_addr.aa_shelf;
    tx_slot = port->port_addr.aa_slot;
    tx_port = port->port_addr.aa_port;
    tx_vci = SHELF_SLOT_PORT_TO_VCI(m(NN_SIG_VCI, tx_shelf, tx_slot,
        tx_port));
    enq.af_type = LMI_CONFIG_ENQ;
    enq.af_version = LMI_VERSION;
    enq.af_my_address = mac->mac_addr;
    resp = lm_build_es_cfg_resp(mac, &enq, &resp_len);
    if (resp == NULL) {
        goto err_exit;
    }
    resp->lmi_type_spec = LMI_STATUS_IND;
    BUILD_ATM_HDR(&atm_hdr, tx_vci);
    BUILD_UCAST_PREFIX(&prefix, tx_shelf, tx_slot, tx_port);

```

197
lm_util.c
-153-

```

ret = lm_send_es_cfg_resp(prefix, atm_hdr, resp, resp_len);
return (ret);

err_exit:
return (ret);
}

lm_send_es_cfg_resp(prefix, atm_hdr, resp, resp_len)
lm_prefix_t prefix;
lm_atm_hdr_t atm_hdr;
lm_es_cfg_resp_t *resp;
{
    int ret;
    SETUP_TCB;

    if (CHK_VB(LM_VB_MSGS)) {
        printf("sending es_cfg_resp, prefix = 0x%x, atm_hdr = 0x%x\r\n",
            prefix, atm_hdr);
    }
    if (resp == NULL) {
        goto err_exit;
    }
    ret = AAL_DataSendNR(&tcg->my_aal_key, resp, resp_len,
        *((tUINT32 *) &prefix), *((tUINT32 *) &atm_hdr));
    return (ret);

err_exit:
    if (CHK_VB(LM_VB_ERRS)) {
        printf("Couldn't send a null aal msg\r\n");
    }
    return (-1);
}

send_mcast_cfg()
{
    int ret;
}

lm_tcb_t *
lm_init()
{
    lm_tcb_t *ret;
    int err_code;
    tUINT32 i;
    int lm_crt_cfg();
    struct wdb_msg *wmsg;
    lm_glbl_cfg_key_t key;

    ret = (lm_tcb_t *) malloc(SIZE_LM_TCB);
    if (ret == NULL)

```

192

lm_util.c
-154-

```

goto err_exit;

SetGlobalP(ret);
bzero(ret, sizeof(*ret));
ret->cur_bid = 0;
ret->my_node = MHW_GetNodeNumber();
ret->my_shelf = MHW_GetShelfNumber();
ret->my_slot = MHW_GetSlotId();
bzero(&ret->port_tmplt, sizeof(ret->port_tmplt));
ret->port_tmplt.aa_type = AAT_PORT;
ret->port_tmplt.aa_country = USA;
ret->port_tmplt.aa_node = ret->my_node;
ret->port_tmplt.aa_shelf = ret->my_shelf;
ret->tmr_blk = TimerInit(1);
LM_INIT_PORT_ADDR(&ret->nac_atm_addr, ret->my_node, ret->my_shelf,
ret->my_slot, 0);
ret->dflt_mtu_size = LM_DFLT_MTU_SIZE;
ret->dflt_num_mcasts = LM_DFLT_NUM_MCASTS;
ret->verbose = LM_VB_ALL;

ret->mac_q = &ret->mac_queue;
ret->port_q = &ret->port_queue;
ret->vlan_q = &ret->vlan_queue;
ret->mv_q = &ret->mv_queue;
ret->pv_q = &ret->pv_queue;
ret->vc_q = &ret->vc_queue;
init_q(ret->mac_q);
init_q(ret->port_q);
init_q(ret->vlan_q);
init_q(ret->mv_q);
init_q(ret->pv_q);
init_q(ret->vc_q);

/* throw away the 0th bid, as per jlb's recommendation */

bits_get_bit(ret->bid_bits, SIZE_BID_BITS);

err_code = AAL_SAP_Create(LM_START_VCI, LM_END_VCI, LM_AAL_SID,
&ret->my_aal_key);
#ifdef UNIX
GetTid(&ret->mytid);
#else
/* ifdef UNIX */
ret->mytid.Generic = TID_LM;
ret->mytid.Instance = LM_INSTANCE;
#endif
/* ifdef UNIX */
GetPid(&ret->mypid);

ret->do_cfg_wrts = TRUE;
key.tag = GLBL_CFG_KEY;
wmsg = wdb_send_fetch_wait(&key, sizeof(key));

```

193

lm_util.c
-155-

```

if (wmsg == NULL || wdb_get_ercode(wmsg) != 0) {
    lm_kludge_data_init();
} else {
    ret->do_cfg_wrts = FALSE;
    wdb_send_startup_queries(lm_crt_cfg);
    ret->do_cfg_wrts = TRUE;
}
SendProxyCheckin(MHW_GetCardType(), MHW_GetSlotId());
return (ret);

err_exit:
    return (NULL);
}

lm_mac_t *
add_mac(mac_addr)
    lm_mac_addr_t *mac_addr;
{
    lm_mac_t *ret;
    qlink_t *link;
    lm_mac_t *tmp;
    SETUP_TCB;

    if (CHK_VB(LM_VB_TERSE)) {
        printf("adding mac addr %s\r\n", sprint_mac_addr(mac_addr));
    }
    mac_addr->aa_type = AAT_MAC;
    ret = FIND_MAC(tcb->mac_q, mac_addr);
    if (ret != NULL)
        return (ret);

    ret = (lm_mac_t *) malloc(SIZE_LM_MAC);
    if (ret == NULL)
        goto err_exit;

    bzero(ret, SIZE_LM_MAC);
    ret->mac_addr = *mac_addr;
    ret->mac_addr.aa_type = AAT_MAC;
    ret->port = NULL;
    ret->mv_q = &ret->mv_queue;
    init_q(ret->mv_q);
    init_qlink(&ret->mac_link, ret);

    ATCH_MAC_MV_Q(tcb->mv_q, ret);
    PUTQ_SORTED_MAC(&ret->mac_link, tcb->mac_q);
    lm_wrt_mac_cfg(ret, NULL);
    return (ret);

err_exit:
    Crash(999, 0, 0);

```

194
lm_util.c
-156-

```

}

lm_port_t *
add_port(port_addr)
    lm_port_addr_t *port_addr;
{
    lm_port_t *ret;
    SETUP_TCB;

    port_addr->aa_type = AAT_PORT;
    port_addr->aa_country = USA;

    if (CHK_VB(LM_VB_TERSE)) {
        printf("adding port %s\r\n", sprint_port_addr(port_addr));
    }
    ret = FIND_PORT(tcb->port_q, port_addr);
    if (ret != NULL)
        return (ret);

    ret = (lm_port_t *) malloc(SIZE_LM_PORT);
    if (ret == NULL)
        goto err_exit;

    bzero(ret, SIZE_LM_PORT);
    ret->port_addr = *port_addr;
    ret->port_addr.aa_type = AAT_PORT;
    ret->port_addr.aa_lanum = 0;
    ret->mac = NULL;
    ret->pv_q = &ret->pv_queue;
    init_q(ret->pv_q);
    init_qlink(&ret->port_link, ret);

    ATCH_PORT_PV_Q(tcb->pv_q, ret);
    PUTQ_SORTED_PORT(&ret->port_link, tcb->port_q);
    lm_wrt_port_cfg(ret);
    return (ret);

err_exit:
    Crash(998, 0, 0);
}

lm_vlan_t *
add_vlan(vlan_id, mtu, num_mcasts, name)
    lm_vlan_id_t vlan_id;
    int mtu;
    int num_mcasts;
    char *name;
{
    lm_vlan_t *ret;
    int i;

```

125
lm_util.c
157

```

SETUP_TCB;

if (CHK_VB(LM_VB_TERSE)) {
    printf("adding vian %d, mtu = %d, num_mcasts = %d, name = %s\r\n",
        vian_id, mtu, num_mcasts, name);
}
ret = FIND_VLAN(tcb->vian_q, vian_id);
if (ret != NULL)
    return (ret);

ret = (lm_vian_t *) malloc(SIZE_LM_VLAN);
if (ret == NULL)
    goto err_exit;

bzero(ret, SIZE_LM_VLAN);
ret->vian_id = vian_id;
ret->mtu_size = mtu;
ret->num_mcasts = 0;
ret->dflt_mlid = LM_MAX_MLID;
strcpy(ret->vian_name, name);
ret->mv_q = &ret->mv_queue;
ret->pv_q = &ret->pv_queue;
ret->vc_q = &ret->vc_queue;
ret->free_vc_q = &ret->free_vc_queue;
init_q(ret->mv_q);
init_q(ret->pv_q);
init_q(ret->vc_q);
init_q(ret->free_vc_q);
init_qlink(&ret->vian_link, ret);

chg_num_vcs(ret, num_mcasts);

ATCH_VLAN_MV_Q(tcb->mv_q, ret);
ATCH_VLAN_PV_Q(tcb->pv_q, ret);
PUTQ_SORTED_VLAN(&ret->vian_link, tcb->vian_q);
lm_chg_vian_cfg(ret, &ret->dflt_mlid, NULL, NULL, NULL);
return (ret);

err_exit:
    Crash(997, 0, 0);
}

lm_vc_t *
get_free_vc(vian)
    lm_vian_t *vian;
{
    lm_vc_t *ret;
    qlink_t *link;

    ret = NULL;

```

196
lm_util.c
-158-

```

link = HEAD_Q(vlan->free_vc_q);
if (link == NULL)
    goto err_exit;

ret = (lm_vc_t *) link->data;
rmq(link);
PUTQ_SORTED_VC(link, vlan->vc_q);
return (ret);

err_exit:
return (NULL);
}

lm_vc_t *
add_vc(vc_addr)
    lm_vc_addr_t *vc_addr;
{
    lm_vc_t *ret;
    lm_vlan_t *vlan;
    SETUP_TCB;

    if (CHK_VB(LM_VB_TERSE)) {
        printf("adding vc %s\r\n", sprint_vc_addr(vc_addr));
    }
    ret = NULL;

    vlan = FIND_VLAN(tcb->vlan_q, vc_addr->vlan_id);
    if (vlan == NULL)
        goto err_exit;

    ret = FIND_VC(vlan->vc_q, vc_addr);
    if (ret != NULL)
        return (ret);

    ret = get_free_vc(vlan);
    if (ret == NULL)
        goto err_exit;

    ret->vc_addr = *vc_addr;
    ret->ref_cnt = 0;
    return (ret);

err_exit:
return (NULL);
}

chg_num_vcs(vlan, num_vcs)
    lm_vlan_t *vlan;
    tINT32 num_vcs;
{

```

197
lm_util.c
-159-

```

int      ret;
int      delta;
int      i;
qlink_t  *link;
qlink_t  *next;
lm_vc_t  *vc;
int      on_vc_q;

ret = 0;
delta = num_vcs - vlan->num_mcasts;
if (delta < 0) {
    on_vc_q = FALSE;
    link = HEAD_Q(vlan->free_vc_q);
    for (i = 0; i > delta; i--) {
        if (link == NULL) {
            if (on_vc_q)
                break;
            link = HEAD_Q(vlan->vc_q);
            on_vc_q = TRUE;
        }
        if (link == NULL)
            break;
        next = link->next;
        vc = (lm_vc_t *) link->data;
        free_vc(vc);
        ret--;
        link = next;
    }
} else if (delta > 0) {
    for (i = 0; i < delta; i++) {
        add_free_vc(vlan);
        ret++;
    }
}
vlan->num_mcasts = num_vcs;
return (ret);
}

lm_vc_t  *
add_free_vc(vlan)
    lm_vlan_t  *vlan;
{
    lm_vc_t  *ret;
    lm_mac_addr_t  mac_addr;
    lm_bld_t  bld;
    SETUP_TCB;

    bld = bits_get_bit(tcb->bld_bits, SIZE_BID_BITS);
    if (bld == -1)
        goto err_exit;
}

```

195
lm_util.c
-160-

```

if (CHK_VB(LM_VB_TERSE)) {
    printf("getting a free mcast vc for vlan %s, bid = %d\r\n",
        sprint_vlan_id(&vlan->vlan_id), bid);
}
ret = (lm_vc_t *) malloc(SIZE_LM_VC);
if (ret == NULL)
    goto err_exit;

bzero(ret, SIZE_LM_VC);
LM_CLR_MAC_ADDR(&mac_addr);
LM_INIT_VC_ADDR(&ret->vc_addr, vlan->vlan_id, &mac_addr);
ret->bid = bid;
ret->vlan = vlan;
init_qlink(&ret->vc_link, ret);
init_qlink(&ret->vlan_link, ret);
PUTQ_SORTED_VC(&ret->vc_link, tcb->vc_q);
PUTQ_SORTED_VC(&ret->vlan_link, vlan->free_vc_q);

return (ret);

err_exit:
    Crash(994, 0, 0);
}

lm_port_vlan_t *
add_pv(port_addr, vlan_id, mlid)
    lm_port_addr_t *port_addr;
    lm_vlan_id_t   vlan_id;
    lm_mlid_t      mlid;
{
    lm_port_vlan_t tmp;
    lm_port_vlan_t *ret;
    lm_port_t      *port;
    lm_vlan_t      *vlan;
    SETUP_TCB;

    if (CHK_VB(LM_VB_TERSE)) {
        printf("adding pv, port_addr = %s, vlan = %d, mlid = %d\r\n",
            sprint_port_addr(port_addr), vlan_id, mlid);
    }
    tmp.vlan_id = vlan_id;
    tmp.port_addr = *port_addr;
    ret = FIND_PV(tcb->pv_q, &tmp);
    if (ret != NULL)
        return (ret);

    ret = (lm_port_vlan_t *) malloc(SIZE_LM_PORT_VLAN);
    if (ret == NULL)
        goto err_exit;

```

199
lm_util.c
-161-

```
bzero(ret, SIZE_LM_PORT_VLAN);
ret->port_addr = *port_addr;
ret->vlan_id = vlan_id;
ret->port = NULL;
ret->vlan = NULL;
ret->mlid = mlid;
init_qlink(&ret->pv_link, ret);
init_qlink(&ret->port_link, ret);
init_qlink(&ret->vlan_link, ret);
```

```
ATCH_PV_VLAN_Q(tcb->vlan_q, ret);
ATCH_PV_PORT_Q(tcb->port_q, ret);
PUTQ_SORTED_PV(&ret->pv_link, tcb->pv_q);
lm_wrt_pv_cfg(ret);
return (ret);
```

```
err_exit:
    Crash(996, 0, 0);
}
```

```
lm_mac_vlan_t *
add_mv(mac_addr, vlan_id, mid, mlid)
    lm_mac_addr_t *mac_addr;
    lm_vlan_id_t vlan_id;
    int mid;
    lm_mlid_t mlid;
```

```
{
    lm_mac_vlan_t tmp;
    lm_mac_vlan_t *ret;
    SETUP_TCB;

    if (CHK_VB(LM_VB_TERSE)) {
        printf("adding mv, mac_addr = %s, vlan = %d, mid = %d, mlid = %d\r\n",
            sprint_mac_addr(mac_addr), vlan_id, mid, mlid);
    }
    tmp.mac_addr = *mac_addr;
    tmp.vlan_id = vlan_id;
    ret = FIND_MV(tcb->mv_q, &tmp);
    if (ret != NULL)
        return (ret);
```

```
ret = (lm_mac_vlan_t *) malloc(SIZE_LM_MAC_VLAN);
if (ret == NULL)
    goto err_exit;
```

```
bzero(ret, SIZE_LM_MAC_VLAN);
ret->mac_addr = *mac_addr;
ret->vlan_id = vlan_id;
ret->mac = NULL;
```

200
lm_util.c
-162-

```

ret->vlan = NULL;
ret->mid = mid;
ret->mlid = mlid;
init_qlink(&ret->mv_link, ret);
init_qlink(&ret->mac_link, ret);
init_qlink(&ret->vlan_link, ret);

ATCH MV VLAN Q(tcb->vlan_q, ret);
ATCH MV MAC Q(tcb->mac_q, ret);
PUTQ SORTED MV(&ret->mv_link, tcb->mv_q);
lm_chg_mv_cfg(ret, &ret->mlid);
return (ret);

err_exit:
    Crash(995, 0, 0);
}

free_tcb(tcb)
    lm_tcb_t    *tcb;
{
    if (tcb != NULL) {
        FREE_VLAN Q(tcb->vlan_q);
        FREE_MAC Q(tcb->mac_q);
        FREE_PORT Q(tcb->port_q);
        FREE_VC Q(tcb->vc_q);
        FREE_MV Q(tcb->mv_q);
        FREE_PV Q(tcb->pv_q);
        lm_rm_glbl_cfg();
        free(tcb);
    }
}

free_mac(mac)
    lm_mac_t    *mac;
{
    SETUP_TCB;

    if (mac != NULL) {
        if (CHK_VB(LM_VB_TERSE)) {
            printf("freeing mac addr = %s\r\n",
                sprint_mac_addr(&mac->mac_addr));
        }
        lm_rm_mac_cfg(mac);
        rmq(&mac->mac_link);
        FREE_MV Q(mac->mv_q);
        lm_send_es_cfg_ind(mac);
        if (mac->port != NULL) {
            mac->port->mac = NULL;
            mac->port = NULL;
        }
    }
}

```